****

COMPUTER SUPPLEMENT #20

****

In this issue:

BUILDING A WORD LIST USING dBASE — PARROT gives directions for generating a dictionary, including pattern searching.

A TYRO'S COMPUTER SOLUTION — Conrad Phillippi relates his experience in solving a totally unknown cipher.

CLASSICAL CRYPTOGRAPHY COURSE — LANAKI is offering an "electronic cryptography course" via electronic mail.

NURSERY RHYME VIGENÈRE — William Corcoran presents a BASIC program to encipher and decipher messages.

DIGITAL SIGNATURES — FIRE-O gives an introduction to digital signatures, and uses an easy-to-follow example.

ENIGMA EXAMPLE — Fauzan Mirza provides an example of a three rotor Enigma encipherment.

WOPADIMA — XERXES III is offering a set of programs to ease the work related to maintaining word dictionaries.

SOME CLASSIC CIPHERS — Sherry Mayo has a brief overview of some basic cipher systems, with examples.

GUTS OF RSA — Francis Litterio reviews the basic math behind public key cryptography.

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]

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 #19 are available from the Editor.

ISSUE DISKS

$5 per disk; specify issue(s), format and density required. All issues presently fit 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:`

`Dan Veeneman`
`PO Box 2442`
`Columbia, Maryland`
`21045-2442, USA.`

`Or via Electronic Mail:`

`dan@decode.com`

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.

Building a Word List Using dBase

PARROT

A word list, or a dictionary, can be a valuable tool when you are trying to solve Aristocrats.

My dictionary comes from three sources. I purchased from Public Brand Software a huge, 3 disk list of words, which in most cases would be sufficient. C Users catalog furnished me with another list along with a spell-checker I've never used. My third source came from the spell-checker of Microsoft Word version 3.0.

The words were imported into dBase, and separated by length into 3 through 22 letter words. Here is the exact dBase source code to build your own dictionary using an ASCII word list:

```
*first create a main dictionary file
.crea dict
```

```
     Field Name  Type      Width  Dec
     ================================
 1   WORD        Character   25
```

press ENTER to finish. This creates a very simple structure of one field containing 25 characters.

Then from the command line, type:

```
.appe from words.txt deli
```

`words.txt` is the ASCII file that is holding your word list. `deli` stands for delimited by, or separated by, and in this case there aren't any separations.

Then go ahead and create 22 more databases with field lengths corresponding with the word length. `dict8.dbf` has my 8 letter words, `dict10.dbf` has my 10 letter long words, and so on. Here is the code to separate the `dict.dbf` into separated word length:

```
store 0 to count
do while .t.
store count+1 to count
use dict
go count
store trim(word) to a
*trim deleted the trailing spaces.
   if len(a)=3
   use dict3
   appe blank
   repla word with a
   loop
   endif
      if len(a)=4
      use dict4
      appe blank
      repla word with a
      loop
      endif
      **And so on until you reach your desired length
 enddo
```

This is a very time-consuming routine. A 386/25 took an hour to separate out a 300,000 word list. I actually did it in three stages, and this comprehensive list is really too long. As a dictionary fails me, I go to the monster list that hasn't let me down yet. SO94, A-14, I used this to find `BRONTEPHOBIA`. My dictionary goes to `dict22`, but I hardly ever use the lengths over eleven.

When you are combining lists from different sources, they must be sorted, and the duplicates weeded out. The different sources must be broken down to word lengths individually. You may end up with, for example, `word8.dbf` and `dict8.dbf`. Here's the code to combine them into one:

```
.use word8
.appe from dict8            (put all the contents of dict8 into word8)
.sort on word to dict8      (sort the combined list back into dict8)
dict8.bf already exists, overwrite it? (Y/N) yes

the write dupes.prg
    use dict8
    do while .t.
    store word to a
    skip
      if word=a
      dele
      endif
   enddo
.pack   (this will write the non-deleted records into dict8)
```

And you can go ahead and delete `word8.dbf`.

Broken down like this, I use the dictionary constantly. If you have a simple pattern word you wish to search for, say `JKMJXXJPN` from SO94 A-11:

```
.use dict9
.copy stru to temp
.use temp
.appe from dict9 from substr(word,1,1)=substr(word,4,1)
    580 records added
(add from dict9 all nine letter words where the first and fourth
 letters are the same.  I got 580).
.dele all from substr(word,4,1)#substr(word,7,1)
    556 records deleted
(delete all words where fourth letter does not equal seventh)
.dele all for substr(word,5,1)#substr(word,6,1)
```

```
     23 records deleted
.pack
      1 record copied
(pack writes all the records not marked for deletion)
.display (your answer) EXCELLENT
```

Pattern words are extracted from the dictionary as shown above, but what about nine letter words without any of the same letters in the word? I keep a special file just for those. Creating the files goes like this, but I will show four letter words to save space:

```
.!copy dict4.dbf to temp4.dbf
.use temp4
.dele all for substr(word,1,1)=substr(word,2,1)
.dele all for substr(word,1,1)=substr(word,3,1)
.dele all for substr(word,1,1)=substr(word,4,1)
.dele all for substr(word,2,1)=substr(word,3,1)
.dele all for substr(word,2,1)=substr(word,4,1)
.dele all for substr(word,3,1)=substr(word,4,1)
.pack
```

This routine creates a special holding file just for four letter words with non-repeating letters, since it is quite time consuming to extract the non-pattern words.

I hope to hear some feedback on this type of an article.

CRYPTOSYSTEMS JOURNAL

After two and a half years, Tony Patti has done it again. The *Cryptosystems Journal, Volume 3*, weighing in with more than 150 pages, is another substantial compilation of software, hardware designs, and current cryptological news.

Tony, now a Director of College Computing and CIO, covers the PEAK cryptosystem, steganography, PGP and Warlock. He gives a detailed description of an improved RANGER device for generating random numbers. He also covers chaos, the NSA, and a variety of other topics.

Contact:

```
Tony Patti
485 Middle Holland Road
Holland, PA  18966
(215) 579-9888
```

A TYRO's COMPUTER SOLUTION OF A TOTALLY UNKNOWN CIPHER

Conrad M. Phillippi

Among my deceased father's papers I discovered a cipher of some 15,000 characters. It had no word spaces and was broken into random length blocks. Being new to the subject with absolutely no idea how to break a cipher I consulted several books. They were mostly anecdotal, and the methods given required some starting clues or prior knowledge of the class of cipher. So with no idea on where I was going, I attacked it by first creating a data file, and then programming a monogram frequency distribution in QuickBasic. This was more random-like than English-like. Digram and trigram distributions gave some interesting results but no leads. Then, studying the text I noted some repetitive letter groups so I wrote a string search program to locate all of them.

This got me interested in certain repeat groups such as `XOEXEOX` which in itself was a blind alley. But this led to a study of the very simplest repeat groups of the form $ABnAB$, where n is a string of irrelevant characters, out to n as large as 30. Frequency plots revealed some clearly non-random peaks at $n = 5$ and $n = 19$, and smaller peaks in the noise at $n = 12$ and $n = 26$. This suggested a periodicity of length 7 and that, in turn, a 7-letter keyword. At about this point I learned of the ACA and wrote to **MICROPOD** for a reference directed at breaking ciphers.

Meanwhile I did monogram distributions of period 7 and got more English-like results. Then I came across Kerckhoff's method for solving the Vigenère cipher and, on a hunch, programmed it.

I was disappointed when it yielded most probable keywords of 7 random letters and was ready to give up, but decided to go the last step, having invested so much time in the project already. Amazingly, plaintext filled the screen!

The very next day **PHOENIX**'s reply arrived. He had promptly solved some sample text I had sent. The cipher turned out to be a diary of sorts. It was stripped of high frequency letters, loaded with `Q`s and `Z`s and full of abbreviations, all carefully planned.

In retrospect, I arrived at the solution by a great deal of groping, some logical analysis up to a point, and then a bit of luck or intuition at the end. The computer was useful for processing this great mass of data. But being able to program was invaluable for following up promptly on questions and ideas as they arose.

Since then I have joined the ACA, have obtained all back issues of the *Computer Supplement*, and have become interested in the interplay between mind and computer in this fascinating subject.

CLASSICAL CRYPTOGRAPHY COURSE

LANAKI

SUMMARY

I can not tell you how much it means to me be President of the ACA. It is an acceptance by ones peers at the highest level. It took nearly 32 years to arrive in this seat. I could not be more proud. I want to put back into the organization some of the trust, knowledge and friendship afforded to me so freely over the years. I intend for this course to be my positive legacy and will work towards that end with enthusiasm.

We are taking a bold step with the first "electronic cryptography course." I hope that we can develop enough material to collate into a book (or notes) that shall be donated to the ACA at no cost. As long as I can underwrite the project, all members would have free access to the ACA 's *Course in Classical Cryptography* for their enjoyment and learning. We work as a team, we share technology as a team, we solve as a team, we succeed as a team.

As of this writing, I have received about fifty five (55) responses from KREWE interested in participating. Here is my preliminary plan of action that has been approved by our EB to commence October 1, 1995:

GOALS

1. ACA correspondence course will act as an "in-depth" review of the various cipher systems. The "how to's" can be explored with an experienced ACA facilitator.

2. Classical cipher systems can be attacked in several different ways with the object of learning the basic tools for cryptanalysis as well as the history.

3. There are lots of entries into cryptograms, so both student and teacher can learn together. We will publish these procedures to build a tool kit.

4. Cipher variants and special cases can be discussed as appropriate. ACA experts will be asked to help in these areas.

5. The historical significance and development of important cipher systems will be covered.

FACILITATOR

Since the class size (as of this writing) is still reasonable at (55), I have volunteered to act as an ACA course facilitator. Fifty (60) concurrent "honor" students is about my limit for the first course. (I also teach Tae Kwon Do three nights per week plus demo team on Saturdays. CCPD has asked me to teach two Rape-Defense courses this winter, so my plate is semi-full.) **LEDGE** has graciously offered his expert help on the Cryptarithms section.

STUDENTS

Students will be asked to classify their classical crypto experience so that responses can be directed appropriately. There will be no tests or unsolvable challenges, just the pure intellectual enjoyment of learning the history, science and recreation of cryptography. We do not pass or fail, we improve our abilities, we become more adept, we laugh together, we grow together. We are self-directed. We attack cryptographic problems. Those who survive will be awarded a ACA Diploma as valued as any degree on your wall.

READING LIST & REFERENCES

A reading list will be sent/published for all students. As the course progresses and we (facilitator/student) find more, then the list will be improved and updated. The reading list will be publicly available. References will be updated frequently by all to improve our practical "tool kit."

NOTES AND ASSIGNMENTS

Classical Cryptography will run for approximately a year and cover most of the ciphers in *ACA and You*. The notes that we generate as facilitator/student become the property of ACA and may be published at the discretion of the Executive Board or Editor of the *Cryptogram*. First rights of refusal of course materials is expressly given to the ACA. Notes, assignments (yes, I give plenty of homework and special projects) will be available on the `ACA-L` or from the facilitator. I have asked **XAMEN EK** to monitor course assignments as 'specials' to be added to member SOL totals.

COURSE EXPENSES

Expenses for course materials are the responsibility of the student. ACA facilitator efforts are complimentary.

TENETS

Some of the greatest "solves" in cryptography have been accomplished by those with little professional experience and with a different approach to offer. All levels of student are in my class, from beginner to PhD and beyond. When we write to each other or use the `ACA-L` list, I would ask that we endeavor to aspire to the tenets of courtesy, integrity and respect. All student questions are not only encouraged but are essential to our collective growth.

EXCEPTIONS AND ACCESS

Although public key cryptography will be discussed, it is not a primary focus of this course. The course is limited to ACA members only. Students agree not to export to the `INTERNET` any ACA materials and to respect the copyrights of the various authors referenced. Those with access via computer and modem are recommended (but not required) to subscribe and use the ACA electronic mailing list, found at

`ACA-L@vm1.nodak.edu`

and the Crypto Drop Box (CDB), run superbly by **NORTH DECODER** and his team. Call Dr. Metzger or E-Mail him at `metzger@rs1.cc.und.nodak.edu` to get details and copies of my recent papers.

Let me know what your interests are so I can plan/direct this course appropriately. You are my customers and I shall do my best to meet your needs. Let ACA help you learn more about cryptography. Enjoy the fun and pain. Persevere.

Best regards,

LANAKI


```
Contact:  (512) 777-2678 Work
          (512) 777-2684 FAX
          (512) 991-3911 Home

E-Mail  75542.1003@compuserve.com
```

NURSERY RHYME VIGENÈRE

William Corcoran

I use "multiloop Vigenère" to express a generic class of substitution ciphers, with the Hagelin system and rotor systems being subsets of the class, along with the oldtime multiloop Vigenère. Everything is expressed in numbers for a character set of C members, together with `mod` C addition for encipherment and subtraction for decipherment, so the classic tableaus for Vigenère, cogs for Hagelin, and wires for rotors, are thrown out the window.

VIGEN_NU.BAS

```
rem This is "Nursery rhyme Vigenere".
rem It enciphers a plain text message or deciphers an enciphered message.
rem Messages are drawn from a set of C=40 characters ( 26 letters of the
rem alphabet, digits 0-9, period, and - or 'or / for showing space between
rem words), and are converted to numbers to form a series of message numbers.
rem Key is made up of N easy-to-remember lines from nursery rhymes and is
rem entered from the key board.  The lines are converted to numbers and
rem corresponding terms are added mod C to produce the enciphering series.
rem The period of the enciphering series is the LCD (least common
rem denominator) of the key line lengths.
rem The enciphering series terms are combined with corresponding terms
rem from the message number series to produce output numbers, which are
rem finally converted back to characters to produce the output message.
rem Plaintext or ciphertext are entered from file.  Output is sent to file
rem as well as to a printer.
rem Obviously, other easy-to-remember lines could be used as well as
rem nursery rhymes.

C = 40

input"What is the message's file name ? ", messname$
lprint"The message is drawn from file: ";messname$
open messname$ for input as #1
  input #1, Msgtxt$
close #1
lprint"Message text as entered:  ";Msgtxt$
L = len(Msgtxt$)   ' number of characters in the message string
lprint"There are ";L;" characters in the message"

lprint
input"How many lines in the key ? ",N
dim lkey$(N)
dim lline(N)
lprint"Key lines are:  "
for i = 1 to N
   print"Enter key line ";i
   input lkey$(i)
   lline(i) = len(lkey$(i))
```

```
   lprint lkey$(i)
next i
lprint"Line lengths are: ";
for i = 1 to N
  lprint lline(i);
next i
lprint
lprint"The enciphering series period will be the LCD of these line lengths."

input"Enter EE if this is an enciphering action, DD if deciphering: ",AAct$
if AAct$ = "EE" then
    z = 1  ' will make enciphering series additive
    print"This is an enciphering action. "
  elseif AAct$ = "DD" then
    z = -1  ' will make enciphering series subtractive
    print"This is a deciphering action.  "
  else
    print"You must choose EE or DD ! "
end if

dim enciph%(L)   ' this will be the enciphering series
dim kkey%(N,L)  '  lines of key converted to numbers
dim outnumb%(L)  '  output numbers will be converted to output text

for i = 1 to N   ' form number array from lines in key
  xx$ = lkey$(i)
  for j = 1 to L
   jj = j mod lline(i)  ' will cycle key line i
   if jj = 0 then jk = lline(i) else jk = j mod lline(i)
   x$ = mid$(xx$,jk,1)
   call charnumb        ' converts key text to number equivalents
   kkey%(i,j) = numb
  next j
next i

lprint
lprint"ENCIPHERING SERIES: "
for i = 1 to L   ' form enciphering series
  y% = 0  '  initialize to start position i
  for j = 1 to N
   y% = y% + kkey%(j,i)      ' increments y% at position i
  next j
  enciph%(i) = y% mod C
  lprint enciph%(i);
next i

lprint
dim txtnumb%(L)  ' convert message text to number equivalents
lprint"Message text converted to numbers: "
for i = 1 to L
  x$ = mid$( Msgtxt$, i,1)
  call charnumb
  txtnumb%(i) = numb
```

```
  lprint numb;
next i

lprint
input"Select a file name for the output message: ",cimsg$
lprint"Output message numbers: "
for i = 1 to L
  xz = z * enciph%(i)  ' z sets add for EE or subtract for DD
  xc = xz + txtnumb%(i)  '  combines textnumb%() and enciph%()
  if xc < 0 then xc = xc + C    ' avoids negative result when deciphering
  x% = xc mod C
  lprint x%;
  outnumb%(i) = x%
next i
lprint

for i = 1 to L
  x% = outnumb%(i)
  call numchar   ' converts output numbers to characters
  charciph$ = charciph$ + ltr$   ' builds output string
next i

open cimsg$ for output as #2
 write #2, charciph$
close #2

lprint"Resulting text: ";charciph$
lprint"This is filed as ";cimsg$

sub charnumb    ' needed to convert characters to numbers
    shared numb, x$

    select case x$

      case "A", "a"
        numb = 1
      case "B", "b"
        numb = 2
      case "C", "c"
        numb = 3
      case "D", "d"
        numb = 5
      case "E", "e"
        numb = 6
      case "F", "f"
        numb = 7
      case "G", "g"
        numb = 8
      case "H", "h"
        numb = 9
      case "I", "i"
        numb = 10
      case "J", "j"
        numb = 11
      case "K", "k"
        numb = 12
      case "L", "l"
        numb = 13
      case "M", "m"
        numb = 15
      case "N", "n"
        numb = 16
      case "O", "o"
        numb = 17
      case "P", "p"
        numb = 18
      case "Q", "q"
        numb = 19
      case "R", "r"
        numb = 20
      case "S", "s"
        numb = 21
```

```
      case "T", "t"
        numb = 22
      case "U", "u"
        numb = 23
      case "V", "v"
        numb = 25
      case "W", "w"
        numb = 26
      case "X", "x"
        numb = 27
      case "Y", "y"
        numb = 28
      case "Z", "z"
        numb = 29
      case "1"
        numb = 30
      case "2"
        numb = 31
      case "3"
        numb = 32
      case "4"
        numb = 33
      case "5"
        numb = 34
      case "6"
        numb = 36
      case "7"
        numb = 37
      case "8"
        numb = 38
      case "9"
        numb = 39
      case "0"
        numb = 35
      case "."
        numb = 0
      case "'"
        numb = 4
      case "/"
        numb = 14
      case "-"
        numb = 24

  end select
end sub

sub numchar   ' convert numbers to their corresponding letters
  shared x%, ltr$

  select case x%

   case 1
    ltr$ = "A"
   case 2
    ltr$ = "B"
   case 3
    ltr$ = "C"
   case 4
    ltr$ = "'"
   case 5
    ltr$ = "D"
   case 6
    ltr$ = "E"
   case 7
    ltr$ = "F"
   case 8
    ltr$ = "G"
   case 9
    ltr$ = "H"
   case 10
    ltr$ = "I"
   case 11
    ltr$ = "J"
   case 12
    ltr$ = "K"
   case 13
    ltr$ = "L"
   case 14
    ltr$ = "/"
   case 15
    ltr$ = "M"
   case 16
    ltr$ = "N"
   case 17
    ltr$ = "O"
   case 18
    ltr$ = "P"
   case 19
    ltr$ = "Q"
   case 20
    ltr$ = "R"
   case 21
    ltr$ = "S"
   case 22
    ltr$ = "T"
```

```
   case 23                                          ltr$ = "3"
    ltr$ = "U"                                     case 33
   case 24                                          ltr$ = "4"
    ltr$ = "-"                                     case 34
   case 25                                          ltr$ = "5"
    ltr$ = "V"                                     case 35
   case 26                                          ltr$ = "0"
    ltr$ = "W"                                     case 36
   case 27                                          ltr$ = "6"
    ltr$ = "X"                                     case 37
   case 28                                          ltr$ = "7"
    ltr$ = "Y"                                     case 38
   case 29                                          ltr$ = "8"
    ltr$ = "Z"                                     case 39
   case 30                                          ltr$ = "9"
    ltr$ = "1"                                     case 0
   case 31                                          ltr$ = "."
    ltr$ = "2"
   case 32

  end select
end sub

end
```

D1_OUT

```
Message text as entered:  A43'JFCY'WNIL-P0XP/RB6'BV/8N1XXPO-S.KQSN1GUQ'8JBTROA
'C4HSQVR8/48F8B7/ME1NR2-KBMQ30/G-WDCR1NN6KZVJW.DAW6GEAEI-UM6O-M18R/.PFC
/K5CBNYW0/KL.KR6LCSAXWJFV7ME6GAD055GHQFQLRKFVLHGUN/.P9.TA.DY0FEC2O6L'H3RJ5PMF
.GHLICZ8RRT6UPMC-K1J9I6/
There are  224  characters in the message
"Key lines are:  "
"jackandjill"
"wentupthehill"
"tofetchapailofwater"
Line lengths are:
 11  13  19
 "The enciphering series period will be the LCD of these line lengths."
"ENCIPHERING SERIES: "
 19  24  26  0  6  37  36  21  34  23  33  37  31  36  4  18  20  34  9  14  39
 26  26  33  19  34  28  0  22  3  6  8  2  6  8  34  28  34  11  8  24  32  17
 39  38  24  30  19  20  39  35  31  22  20  11  39  4  3  11  17  28  36  24
 32  27  17  18  27  38  12  26  24  15  39  25  3  8  1  34  15  10  26  8
 24  11  20  29  35  38  21  2  39  29  38  7  16  5  2  28  39  13  22  26
 32  3  21  0  9  3  17  34  32  15  1  33  6  25  3  38  32  4  35  37  26
 31  20  2  22  10  24  21  25  7  5  3  37  29  7  14  29  21  4  27  7  20
 36  1  10  35  9  26  36  4  0  31  20  11  21  26  39  6  30  2  35  36  27
 37  17  7  33  2  3  10  0  37  17  23  16  13  0  15  39  4  34  33  21  37
 11  29  32  0  27  33  24  6  39  28  30  11  25  31  7  27  29  0  22  5
```

```
 32  19  39  34  32  1  1  1  23  18  37  24  36  37  4  16  14
 Message text converted to numbers:
 1  33  32  4  11  7  3  28  4  26  16  10  13  24  18  35  27  18  14  20  2
 36  4  2  25  14  38  16  30  27  27  18  17  24  21  0  12  19  21  16  30  8
 23  19  4  38  11  2  22  20  17  1  4  3  33  9  21  19  25  20  38  14  33
 38  7  38  2  37  14  15  6  30  16  20  31  24  12  2  15  19  32  35  14  8
 24  26  5  3  20  30  16  16  36  12  29  25  11  26  0  5  1  26  36  8  6
 1  6  10  24  23  15  36  17  24  15  30  38  20  14  0  18  7  3  14  12  34
 3  2  16  28  26  35  14  12  13  0  12  20  36  13  3  21  1  27  26  11  7
 25  37  15  6  36  8  1  5  35  34  34  8  9  19  7  19  13  20  12  7  25
 13  9  8  23  16  14  0  18  39  0  22  1  0  5  28  35  7  6  3  31  17  36
 13  4  9  32  20  11  34  18  15  7  0  8  9  13  10  3  29  38  20  20  22
 36  23  18  15  3  24  12  30  11  39  10  36  14
 Output message numbers:
 22  9  6  4  5  10  7  7  10  3  23  13  22  28  14  17  7  24  5  6  3  10
 18  9  6  20  10  16  8  24  21  10  15  18  13  6  24  25  10  8  6  16  6
 20  6  14  21  23  2  21  22  10  22  23  22  10  17  16  14  3  10  18  9  6
 20  21  24  10  16  3  20  6  1  21  6  21  4  1  21  4  22  9  6  24  13  6
 16  8  22  9  14  17  7  14  22  9  6  24  12  6  28  4  10  16  3  20  6  1
 21  6  21  4  2  23  22  24  13  17  16  8  14  12  6  28  21  14  1  20  6
 4  5  10  7  7  10  3  23  13  22  24  22  17  14  20  6  15  6  15  2  6  20
 0  4  1  14  15  23  13  22  10  13  17  17  18  24  25  10  8  6  16  6  20
 6  14  3  1  16  24  9  1  25  6  24  1  14  25  6  20  28  4  13  17  16  8
 14  12  6  28  4  22  9  1  22  24  10  21  24  6  1  21  28  4  22  17  14
 20  6  15  6  15  2  6  20  0
 "Resulting text: ","THE'DIFFICULTY/OF-DECIPHERING-SIMPLE-VIGENERE/SUBSTITUTION
/CIPHERS-INCREASES'AS'THE-LENGTH/OF/THE-KEY'INCREASES'BUT-LONG/KEYS/ARE'
DIFFICULT-TO/REMEMBER.'A/MULTILOOP-VIGENERE/CAN-HAVE-A/VERY'LONG/KEY'THAT-IS-
EASY'TO/REMEMBER."
```

E1_OUT

```
Message text as entered:  The'difficulty/of-deciphering-simple-Vigenere
/substitution/ciphers-increases'as'the-length/of/the-key'increases'but
-long/keys/are'difficult-to/remember.'A/multiloop-Vigenere/can-have-a
/very'long/key'that-is-easy'to/remember.
There are  224  characters in the message
Key lines are:
jackandjill
wentupthehill
tofetchapailofwater
Line lengths are:
 11  13  19
"The enciphering series period will be the LCD of these line lengths."
ENCIPHERING SERIES:
 19  24  26  0  6  37  36  21  34  23  33  37  31  36  4  18  20  34  9  14
 39  26  26  33  19  34  28  0  22  3  6  8  2  6  8  34  28  34  11  8  24
 32  17  39  38  24  30  19  20  39  35  31  22  20  11  39  4  3  11  17
 28  36  24  32  27  17  18  27  38  12  26  24  15  39  25  3  8  1  34  15
 10  26  8  24  11  20  29  35  38  21  2  39  29  38  7  16  5  2  28  39
```

```
 13  22  26  32  3  21  0  9  3  17  34  32  15  1  33  6  25  3  38  32  4
 35  37  26  31  20  2  22  10  24  21  25  7  5  3  37  29  7  14  29  21
 4  27  7  20  36  1  10  35  9  26  36  4  0  31  20  11  21  26  39  6  30
 2  35  36  27  37  17  7  33  2  3  10  0  37  17  23  16  13  0  15  39  4
 34  33  21  37  11  29  32  0  27  33  24  6  39  28  30  11  25  31  7  27
 29  0  22  5  32  19  39  34  32  1  1  1  23  18  37  24  36  37  4  16  14
 Message text converted to numbers:
 22  9  6  4  5  10  7  7  10  3  23  13  22  28  14  17  7  24  5  6  3  10
 18  9  6  20  10  16  8  24  21  10  15  18  13  6  24  25  10  8  6  16  6
 20  6  14  21  23  2  21  22  10  22  23  22  10  17  16  14  3  10  18  9
 6  20  21  24  10  16  3  20  6  1  21  6  21  4  1  21  4  22  9  6  24  13
 6  16  8  22  9  14  17  7  14  22  9  6  24  12  6  28  4  10  16  3  20  6
 1  21  6  21  4  2  23  22  24  13  17  16  8  14  12  6  28  21  14  1  20
 6  4  5  10  7  7  10  3  23  13  22  24  22  17  14  20  6  15  6  15  2  6
 20  0  4  1  14  15  23  13  22  10  13  17  17  18  24  25  10  8  6  16  6
 20  6  14  3  1  16  24  9  1  25  6  24  1  14  25  6  20  28  4  13  17
 16  8  14  12  6  28  4  22  9  1  22  24  10  21  24  6  1  21  28  4  22
 17  14  20  6  15  6  15  2  6  20  0
 Output message numbers:
 1  33  32  4  11  7  3  28  4  26  16  10  13  24  18  35  27  18  14  20  2
 36  4  2  25  14  38  16  30  27  27  18  17  24  21  0  12  19  21  16  30
 8  23  19  4  38  11  2  22  20  17  1  4  3  33  9  21  19  25  20  38  14
 33  38  7  38  2  37  14  15  6  30  16  20  31  24  12  2  15  19  32  35
 14  8  24  26  5  3  20  30  16  16  36  12  29  25  11  26  0  5  1  26  36
 8  6  1  6  10  24  23  15  36  17  24  15  30  38  20  14  0  18  7  3  14
 12  34  3  2  16  28  26  35  14  12  13  0  12  20  36  13  3  21  1  27
 26  11  7  25  37  15  6  36  8  1  5  35  34  34  8  9  19  7  19  13  20
 12  7  25  13  9  8  23  16  14  0  18  39  0  22  1  0  5  28  35  7  6  3
 31  17  36  13  4  9  32  20  11  34  18  15  7  0  8  9  13  10  3  29  38
 20  20  22  36  23  18  15  3  24  12  30  11  39  10  36  14
 "Resulting text: ","A43'JFCY'WNIL-P0XP/RB6'BV/8N1XXP0-S.KQSN1GUQ'8JBTR0A
'C4HSQVR8/48F8B7/ME1NR2-KBMQ30/G-WDCR1NN6KZVJW.DAW6GEAEI-UM6O-M18R/.PFC
/K5CBNYW0/KL.KR6LCSAXWJFV7ME6GAD055GHQFQLRKFVLHGUN/.P9.TA.DY0FEC2O6L'H3RJ5PMF
.GHLICZ8RRT6UPMC-K1J9I6/"
```

SECRET WRITING EQUIPMENT

Crayola has come out with a product called *Secret Writers Markers*™ , advertised as

> Write a secret message with the invisible writer, use the color decoder to reveal it!

The package contains two "invisible writers" and six "color decoders" that allow the users to create and later develop hidden messages. This may be just the thing to introduce children to cryptography and "secret writing."

I found a set in the office supply section of a local discount store. Also, Crayola maintains a question and comment line at 1-800-CRAYOLA, available weekdays 9 am to 4 pm Eastern time.

DIGITAL SIGNATURES

Fire-O

The two greatest developments in cryptography over the past 15 years have been Public Key Cryptography and Digital Signatures. These two developments go hand-in-glove. Public key cryptography is most valuable precisely because it provides digital signatures, and digital signatures depend on the methods of public key cryptography.

A digital signature is analogous to a handwritten signature. A hand signature on a piece of paper, such as a will or contract, assures anyone who sees that document that it is genuine. A digital signature does the same thing for a message sent electronically through a message system, such as a computer network.

To be fully useful, the digital signature must do three things. First, it must assure the receiver that the message comes from the indicated sender. Second, it must assure the receiver that the message has not been altered in any way. Third, the signature must prove to an impartial third party that the message is genuine, and comes from the purported sender.

Conventional secret key cryptography can satisfy the first two requirements. If the sender and receiver have decided on a secure cryptographic method, and agreed on a unique key, then any message encrypted by that method using that key must be genuine.

The third requirement, however, is not easy to achieve with conventional cryptography. Suppose that the receiver shows a judge a message, and says that it came from a certain sender. How is the judge to know if sender and receiver had such an arrangement?

It could be that the receiver made up both the message and the key. Also, the judge will now be able to read all past and future messages between this sender and receiver, and create fake messages. A system of digital signatures must rely solely on public information, and not on private methods or keys.

The need to verify the validity of messages has been solved in the computer field for many years. In the early days of computers, data transmission was somewhat unreliable. So it became a convention to add hash values or checksums to messages. A very simple example is for the sender to add all of the characters in a message, and append the sum to the end. The receiver does the same. If the message has been garbled, the sums won't match.

That simple method words fine for accidental changes to a message, but it provides little protection against deliberate tampering. A person altering the message could either change the checksum, or make two changes to the message so that the checksum is unchanged.

There are really two problems here. The first is that the interloper knows what the checksum is, and can change the checksum, and the second is that it is easy to figure out how to change the message so the checksum doesn't change. So even if we protected the checksum, say by enciphering it, or by sending it separately from the message, we cannot guarantee the validity of the message.

This problem is solved by using what is called a one-way hash function. This is a function that is easy to compute, but difficult to invert or reverse. Such functions are very hard to find, but a few are known. Probably the simplest such function is just squaring the number.

Suppose that N is a very large number, say 130 decimal digits or longer, and N is the product of two large primes p and q, which are at least 50 digits each. (A prime number is an integer which is not the product of two smaller integers. For example, 2, 3, 5, and 7 are primes, but 9 is not, since it is 3×3.)

It is very difficult to factor such a large number, that is to find p and q. It could take many years, even on today's fastest computers. If A is any number, let A mod N denote the residue or remainder when A is divided by N. That is, we ignore the quotient, and just keep the remainder. For example, 18 mod 6 is 0, 19 mod 6 is 1, 20 mod 6 is 2, and so forth.

If A is an unknown number, and we know the value of p and q, then if we are given the value of A^2 mod N we can easily find A. There is a formula that gives A directly from A^2 mod N. But if we don't know the factors of N, then there is no way to find A from A^2 mod N. It is easy for anyone to compute A^2 mod N, just multiply A times A, divide by N, and keep the remainder. But, given just A^2 mod N and N, it is very difficult to find A. So this function A^2 mod N is a good one-way hash function.

Given a one-way hash, it is now easy to get a digital signature. Any message may be treated as a numerical value in a variety of ways. For example, the characters of the message may be represented as two-digit decimal numbers, $A = 01$, $B = 02$, ... $Z = 26$. Then the message HELP would have the numeric value 08051216, or 8051216, dropping the initial zero.

Suppose, now, that every sender in the message network has chosen one of these giant numbers, N, as a public key, and that these keys are all publicly known. For example, there may be a book published giving everyone's key, or there may be a public computer file that contains everyone's name and key.

If a given sender wants to send and sign the message `HELP`, all that is needed is to find the number S (the signature) whose square is 8051216. Since the sender knows the factors of N, this is easy. Then the sender transmits 8051216 followed by the signature, S. The receiver can square the signature, that is compute S^2 mod N, and compare it to 8051216. If it matches, the message must be genuine and must come from the sender, because nobody else could have computed the signature.

Another person could not change the message, because that would require finding a different signature. Since the sender's public key is known to everyone, the receiver could show the message and signature to a third party, and that party could verify independently that the message was genuine.

A digital signature is far more difficult to fake than a written signature. It provides a secure way of transmitting contracts, cash or stock transfers, judicial decisions, or any message where reliable communications are critical.

ACA COMPUTER BULLETIN BOARD UPDATE

All members of Krewe are welcome to use the ACA bulletin board system, `Decode`, for electronic mail to the Internet. It is available 24 hours a day at +1 410 730 6734.

Each user will automatically gain an Internet address of the form $<$ *user* $>$`@decode.com`, and may correspond via e-mail to members of the Krewe and other Internet users.

The `FILES` section also contains various ACA and cryptographic-related files and programs, as well as an assortment of other topics.

3-ROTOR ENIGMA ENCIPHERMENT EXAMPLE

Fauzan Mirza

How the Enigma works:

```
               P  1  2  3  R
key ---------->|->|->|->|->|
               |  |  |  |  |
output <------|<-|<-|<-|<-|
```

```
P: Patchpanel
1: Rotor 1
2: Rotor 2
3: Rotor 3
R: Reflecting rotor
```

Current passes through the rotors : `1 2 3 R 3 2 1`

The order that the three rotors operate can be changed. The patchpanel operates on the character before and after it has been processed by the rotors.

The reflecting rotor is like the other rotors except that the mapping is symmetrical.

MACHINE PREPARATION

```
  Rotor wirings:
Input:  ABCDEFGHIJKLMNOPQRSTUVWXYZ
1:      EKMFLGDQVZNTOWYHXUSPAIBRCJ
2:      AJDKSIRUXBLHWTMCQGZNPYFVOE
3:      BDFHJLCPRTXVZNYEIWGAKMUSQO
4:      ESOVPZJAYQUIRHXLNFTGKDCMWB
5:      VZBRGITYUPSDNHLXAWMJQOFECK
R:      YRUHQSLDPXNGOKMIEBFZCWVJAT
```

1. Select three out of the five rotors and arrange them in an order.
2. Select the rotor ring settings and starting positions.
3. Connect pairs of letters on the plugboard.

These are the key to the enciphered message.

Example:

```
Rotor order:              3 1 2
Alphabet ring setting:    W X T
Rotor starting positions: A W E
Plugboard:    (AM) (TE)
```

ENCIPHERMENT

Get input character to encipher.

Example: Input character is `T`. It will be manipulated during the encryption process, and we shall always refer to the current form of the letter as the input character (even though it is unlikely to resemble the original letter).

Each rotor has a physical notch on it which determines when the next rotor is stepped up. The following table shows which letter is visible in the rotor window when the notch is in the engaged position:

```
Rotor notches:
1: Q
2: E
3: V
4: J
5: Z
```

Remember that the notch is physical and will step up regardless of which rotor order is being used.

Step up the first rotor. If the first rotor has reached its notch then step up the second rotor.

Example: When the first rotor, rotor 3, reaches `V` then the second rotor, rotor 1, steps up to `X`.

If the second rotor just reached its notch after being stepped up then at the next round, step both the second and third rotors up once.

Example:

```
Rotor positions: B W E
```

PATCHPANEL

The patchpanel modifies the input character depending on the plugboard connections. It operates on the character before and after processing by the rotors.

Simply lookup the plugboard pairs to see if the input character is in any of the pairs. If it is then substitute the input character for the other character in the pair.

Example:

```
Plugboard connections are (AM) (TE)
Input character is a 'T'.
Change it to an 'E'.
```

Do the following for each of the rotors in rotor order.

1. Rotate the character by the current rotor position.
2. Shift the input alphabet until the `A` of the "Forward" alphabet is aligned with the current rotor position letter in the "Backward" alphabet.
3. Translate the input character.

```
Forward:     ABCDEFGHIJKLMNOPQRSTUVWXYZ
Backward:   ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Example: First rotor (rotor 3) is `B` (Align `A` with `B`, shown above). The input character is `E` which translates to `F` (forward). This is basically a rotate operation on the input character. (it is being rotated depending on the current rotor position).

Align the following output table so that the `A` in the "Backward" alphabet is below the rotor ring setting in the "Forward" alphabet.

```
Forward:    ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ
Backward:                            ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Look up the letter, which corresponds to the input character in the above table, in the rotor wirings and find its corresponding forward letter in the above table. The result is the new input character.

Example: Ring setting for first rotor (rotor 3) is `W`. Aligning the letter `A` in the "Backward" alphabet with rotor ring setting `W` in the "Forward" alphabet, the letter corresponding to the input character `F` is `J` (forward). The rotor wiring for input `J` gives output `T`. The letter corresponding to `T` in the above table is `P` (backward). Input character becomes `P`.

Rotate the character back by the current rotor position. We are reversing the rotation made before the rotor wiring lookup, which is why it is rotated backwards. The table to rotate it forward is used to translate the input character, by looking up the input character the table backwards.

Example: First rotor (rotor 3) is `B` (`A` aligned with `B`). The input character is `P` which translates to `O` (backward).

Repeat this process for the next rotor. When processed with the three rotors, translate the

input character through the reflecting rotor. This is a simple lookup.

Example: By this stage, the input character would be an `M`. The reflecting rotor translates this into a `O`.

Now the input character is processed by the rotors in reverse order. This is exactly the same as when processed by the rotors as before, except that the lookup in the rotor wirings is reversed. The rotor order is also reversed.

Example: The input character `O` is passed back to the third rotor (rotor 2) and goes on to the second rotor (rotor 1) and first rotor (rotor 3) before being modified by the patchpanel. The output from the patchpanel is the final result.

The input to the last rotor (rotor 3) should be `G`. This is rotated to the letter `H`. Align the `A` in the "Backward" alphabet with the rotor ring setting in the "Forward" alphabet. (Rotor ring setting is `W`). In the "Forward" alphabet, the character corresponding to the input character `H` is `L`. The character corresponding to the letter `L` in the rotor wirings (reverse lookup) is `F`. In the "Backward" alphabet, the character corresponding to the input character `F` is `B`. The `B` is rotated back to `A`. Output is `A`.

The patchpanel operates on the output from the reversed rotors. Once again, this is simply exchanging of letter pairs if the letter is one of a pair on the plugboard.

Example: Input character is `A`. Plug (AM) changes this into an `M`.

The output from the patchpanel is the final, enciphered, letter. Carry on enciphering message, one letter at a time.

Example: Enciphering `THISISATEST` using the settings described at the start.

Input	P	3	1	2	R	2	1	3	P	Output
T	E	[FJTPO]	[KNWTX]	[BIXQM]	(MO)	[SZSLH]	[DGFCG]	[HLFBA]	A	M
H	H	[JNNJH]	[DGDAE]	[IPCVR]	(RB)	[FMOHD]	[ZCYVZ]	[BFCYW]	W	W
I	I	[LPEAX]	[TWBYC]	[GNTMI]	(IP)	[TAATP]	[LOMJN]	[QUWSP]	P	P
S	S	[WABXT]	[PSSPT]	[XESLH]	(HD)	[HOYRN]	[JMCZD]	[HLFBX]	X	X
I	I	[NRWSN]	[JMOLP]	[TAATP]	(PI)	[MTNGC]	[YBWTX]	[CGSOJ]	J	J
S	S	[YCFBV]	[RUAXB]	[FMWPL]	(LG)	[KRGZV]	[RUROS]	[YCGCW]	W	W
A	M	[TXSOH]	[DGDAE]	[IPCVR]	(RB)	[FMOHD]	[ZCYVZ]	[GKUQJ]	J	J
T	E	[MQIEW]	[SVIFJ]	[NUPIE]	(EQ)	[UBJCY]	[UXQNR]	[ZDBXP]	P	P
E	T	[CGCYP]	[LOYVZ]	[DKLEA]	(AY)	[CJBUQ]	[MPTQU]	[DHDZQ]	Q	Q
S	S	[CGCYO]	[KNWTX]	[BIXQM]	(MO)	[SZSLH]	[DGFCG]	[QUWSI]	I	I
T	E	[PTAWL]	[HKNKO]	[SZEXT]	(TZ)	[DKDWS]	[ORXUY]	[JNNJY]	Y	Y

WOPADIMA

XERXES III

WOPADIMA General Description

The WOPADIMA system is a system of computer programs for WOrd PAttern DIctionary MAintenance. It runs on IBM and compatible computers using the DOS operating system. It can create and update a set of computer files of word patterns. It can also make files in a format that is ready for computer listing to produce a notebook version of the computer's file of word patterns. The system does not include any provision for deleting unwanted records from the files. It was assumed that the user would have some kind of editor program, such as KEDIT, to view the files and extract records from the files. That editor can also be used to delete records from the files.

The WOPADIMA system can handle words up to 24 letters long. The word pattern records for each word length are kept in separate files. The system will handle words containing apostrophies and treats the apostrophy the same as a letter. No other punctuation marks are recognized by the system. Words with apostrophies and words of the same length without apostrophies are kept in separate files.

Words may be input into the system from any machine readable source that contains words in ASCII. Repetition of words in the input and inclusion of words that are already in the system's files are no problem. The system takes these in stride and does not put duplicate records into its files.

The WOPADIMA system consists of seven programs. Four of these are necessary to create and maintain the computer files of word pattern records. Another one is necessary to produce the notebook version of the computer's word pattern files. The other two are convenience programs that are never necessary but might be wanted occasionally.

All input to the system is processed by `GENWDINP` (GENeral WorD INPut). The input must be in ASCII form and may be in any directory. The user designates the name of the input file including the path if the file is not in the default directory. The user may enter any valid name for the output file including the path if that file is not in the default directory. If the designated output file does not exist it will be created. If it does exist the generated output will be concatenated onto it.

Each word in the input file will be analyzed individually and will create a record in the output file. The output record will be in the long format consisting of the word length in a 2-byte ASCII field, a blank, the word pattern, a blank, and the word in capital letters. Each record is delimited by the machine code for carriage return-line feed (`0x0D 0x0A`) which is invisible to the user when viewing the file. The word `hello` in any combination of capital and lower case letters will produce the output record `05 01223 HELLO` delimited by the `CrLf`.

`DIVVYUP` takes the output records from `GENWDINP` and divides them up into separate files each containing words of only one length and all with or all without apostrophies. This could, conceptually, produce 48 output files but much smaller numbers are the norm. The user does not choose the names for the output files. `DIVVYUP` places all its output files in the default directory and names them `WORDnn.ACT` or `APOSnn.ACT`. `WORD` indicates a file of words without apostrophies and `APOS` indicates a file of words with apostrophies. `nn` is the ASCII form of the length of the words in that file and `.ACT` designates the file as activity. If any of these files exist at the time it is called the new output will be concatenated onto it. The rigid naming and directory location of the output files does not decrease the user's flexibility in using the system. The program can be invoked with its path if it is not in the default directory and the input file name can include the path if it is not in the default directory. The output

files can be renamed if the assigned names are not acceptable.

The program will create both a `WORD` file and an `APOS` file for each word length that it processes even if there are no records in one of these files. Output files that contain no records can be erased.

Output records from `DIVVYUP` are in the short format, i.e., the record `05 01223 HELLO` in the input will produce the record `1223 HELLO` in the output file `WORD05.ACT`. There is no loss of information because all records in that file are five letter words and all patterns begin with a `0`.

The input file for `DIVVYUP` may be in any order but the program will run *much* faster if all records of one word length are grouped together. It is customary to sort the input file into ascending order because the output files will then be in ascending order (unless they were concatenated) and they must be in ascending order for all subsequent processing.

`WHATSNU` (WHAT iS New) takes an output file from `DIVVYUP`, compares it with the corresponding file in the computer's word pattern dictionary, and produces an output file containing only those records that are not already in the dictionary. Neither input file is altered. Both input files must be short format records for the same word length and both must be in ascending ASCII sort. The output file will not contain any duplicate records.

The output file from `WHATSNU` is the shortest possible list of the activity for that word pattern file. It is the easiest place to do the necessary editing. Words that are not wanted in the main dictionary, data entry errors, etc. should be deleted from this activity file before it is merged into the word pattern dictionary.

`MERGWDS` (MERGe WorDS) will merge an edited activity file with a word pattern dictionary file. Both input files must be for the same word length and both must be in ascending ASCII order. Neither input file is altered. `MERGWDS` will create an output file containing all of the records in both input files. The output file will be in ascending order. The designated output file must not exist when it is invoked. The merged output file can be copied into the word pattern dictionary, which will replace the former dictionary file, after it has been accepted. That completes updating the word pattern dictionary in its computer file form.

`PRNTDWBU` (PRiNT Dictionary With Back Up) will read a word pattern dictionary file and create two output files in notebook format. One file contains the odd numbered pages and the other one contains the even numbered pages. This simplifies the task of printing the notebook pages on both sides of the sheets. The input file must be in the short format and must contain records of only one word length in ascending order. The user can choose any path and filename for the output files but *must not* include an extension. `PRNTDWBU` adds the extension `.ODD` to the file of odd numbered pages and `.EVE` to the file of even numbered pages. The output file with the extension `.ODD` is copied to the printer first. The paper is then removed from the printer and reloaded so that the first page to be printed is the back of page 1, the next the back of page 3, etc. The output file with the extension `.EVE` is then copied to the printer.

These five programs are all that are necessary in the WOPADIMA system. `WHATSNU` and `MERGWDS` each eliminate duplications in the records so it is never necessary to eliminate them any other way. However, some users might want to eliminate duplication from their activity files. Two programs are included to do that. Both programs require the input files to contain records of only one word length sorted in ascending order. Both programs do not alter the input file but do create an output file in the same format as the input but without duplications.

`NOSHDUPS` (NO SHort DUPlicateS) works for the short format files that are used everywhere after `DIVVYUP`. These files already contain only

one word length so the only requirement is that the file be in the proper sort.

`NOLODUPS` (NO LOng DUPlicateS) works for the long format files that exist between `GENWDINP` and `DIVVYUP` but the output file from `GENWDINP` nearly always contains words of more than one length. Single word length files would have to be extracted from the mixed word length file before `NOLODUPS` could be used.

The word pattern dictionary files in computer format are the main repository for the word patterns. They are also the source of word pattern information when a person is using the computer as an aid for manual solution of aristocrat cryptograms. Those files always contain the most complete and up-to-date information. Most users will sometimes work at cryptogram solution using a totally manual approach without the computer's help. The printed notebook form of the files serves this purpose. The printed notebook is usually not complete and up-to-date because it is not reprinted every time the computer files are updated.

The WOPADIMA system was written in Assembler and assembled with `MASM4` for use on IBM and compatible computers using the DOS operating system.

The system will allow the user to:

1. Create a set of word pattern dictionary files.
2. Add new words to an existing dictionary.
3. Print any file(s) in the dictionary in notebook format printing on both sides of the pages.

The system will handle words up to 24 letters long. It does not recognize any punctuation except apostrophies. Words that have apostrophies are kept in a separate set of files from the words that do not. The apostrophe is counted and treated the same as a letter. The system treats as a word any collection of letters and apostrophies that occur between two characters that are neither letters nor apostrophies. It does not test the "word" for validity but it does make provision for the user to screen the input lists before they are merged into the dictionary.

The system does not, at present, provide any assistance in applying the dictionary to the solution of cryptograms.

Machine readable copies of the system are available to any ACA members who want them. I ask those people who want it on 3 1/2" disks to send me a blank HD disk or 85 cents. (Anyone requiring DD disks should send 2 blank DD's.) Anyone who is willing to accept it on 5 1/4" disks only needs to tell me that he wants it because I have a surplus of those disks. (If they do not tell me otherwise I will assume that they can use either DD or HD disks.)

The disks will contain the following files.

1. A description of the system.
2. A tutorial covering the use of the system.
3. The programs in `.EXE` form.
4. The source code (`.ASM` form) for the programs.
5. A recent copy of my word pattern dictionary.
6. A short discussion on using the word pattern dictionary.

`XERXES III`
`Harold X. Brown`
`PO Box 20631`
`Indianapolis, IN  46220-0631`

SOME CLASSICAL CIPHERS

Sherry Mayo

The Caesar Cipher

This is an extremely simple cipher which is attributed to Caesar. It is a substitution cipher, where each letter of the alphabet is replaced by another letter of the alphabet as follows:

```
 plaintext alphabet: a b c d e f g h i j k l m n o p q r s t u v w x y z
ciphertext alphabet: d e f g h i j k l m n o p q r s t u v w x y z a b c
```

It is clear from the table above that each letter has been replaced by the letter three places further along in the alphabet. At the end of the alphabet we just go back to the start, thus `x` is replaced by `a`, `y` by `b`, and `z` by `c`.

Although the relative shift between the plaintext and ciphertext alphabets in the example above is 3, it could be any number between 1 and 25. The shift number (i.e. 3 in this case) is the key to the cipher.

Using the substitution above, the plaintext message `london calling moscow` is enciphered as follows: (It is customary to omit the spaces in ciphertext.)

```
plaintext: london calling moscow
ciphertext: orqgrqfdooljprvfrz
```

To convert the encrypted message back to plaintext, simply reverse the encryption process. If the key, n, is known (3 in this case) simply replace each letter with the letter n places before it in the alphabet.

Weaknesses

Even if the key is unknown, it is not hard to decrypt a message encrypted using the caesar cipher. The simplest technique would be to try all 25 possible replacement alphabets until the resulting decrypted message made sense.

Frequency analysis of letter occurences could also be used to find the key. The most frequently occurring letter is likely to correspond to `e`, which is the most frequently occurring letter in the english language (provided the original message was in english!). Thus if `r` is the most frequent letter in the ciphertext then the key is likely to be 13. Note that frequency analysis needs a reasonable amount of ciphertext to be at all reliable.

The Augustus Cipher

The cipher is closely related to the Vigenère cipher, and is attributed (possibly erroneously) to Emperor Augustus. The story goes that he used a passage from Homer as the key to encrypt his messages. The key in this case is equal to the length of the plaintext — you simply use as much of the key text as is required.

To encrypt the mth letter of the plaintext, select the mth letter of the key text; the position

of this letter in the alphabet determines the shift for the plaintext letter. In other words, if the mth plaintext letter is `o` and the mth key text letter is `c`, the shift is 3 because `c` is the 3rd letter in the alphabet, and thus `o` is replaced by the `r`, which is 3 places further along in the alphabet. If in shifting a letter of plaintext you "run out" of alphabet, you start again at `a`, e.g. the plaintext letter `w` encrypted by the key letter `f` (shift = 6) would result in the ciphertext letter `c`.

Here is an example of the plaintext `London calling Moscow with urgent message` encrypted using the words of Hamlet's famous soliloquy:

```
 plaintext: l  o  n  d  o  n  c  a  l  l  i  n  g  m  o  s  c  o
  key text: t  o  b  e  o  r  n  o  t  t  o  b  e  t  h  a  t  i
     shift: 20 15  2  5 15 18 14 15 20 20 15  2  5 20  8  1 20  9
ciphertext: f  d  o  i  d  f  q  p  f  f  x  o  l  g  w  t  w  x

 plaintext: w  w  i  t  h  u  r  g  e  n  t  m  e  s  s  a  g  e
  key text: s  t  h  e  q  u  e  s  t  i  o  n  w  h  e  t  h  e
     shift: 19 20  8  5 17 21  5 19 20  9 15 14 23  8  5 20  8  5
ciphertext: p  q  q  y  y  p  w  z  y  w  i  a  b  a  x  u  o  j
```

This gives the final ciphertext of:

`fdoidfqpffxolgwtwxpqqyypwzywiabaxuoj`

The Vigenère tableau can be used to assign letters as for the Vigenère cipher. the difference in this case is that the sequence of substituted alphabets doesn't repeat but is instead determined by the keytext.

Deciphering the text is just a matter of reversing the process described above using the same key text. e.g., for the first letter of the ciphertext shown above, `f`, the first letter of the keytext, `t` determines the shift of 20. This ciphertext letter is then replaced by the letter 20 places before it in the alphabet, `l`, and so on for the rest of the message.

Weaknesses

Since the sequence of alphabets doesn't repeat, frequency analysis cannot be used to help break this cipher. However, since the key is a piece of text dictionary type attacks can be used to find the key word by word. Once you have a few words you may even be able to identify the text that was used, thus speeding up the process.

The Playfair Cipher

The Playfair cipher uses a keyword to scramble an alphabet which is then written into a 5×5 array (the letters `i` and `j` are treated as the same letter).

For the keyword `Inctatus` one way of doing this is as follows

```
INCTA
USBDE
FGHKL
```

```
MOPQR
VWXYZ
```

Write out the keyword into the array (omitting duplicated letters) followed by the remaining letters of the alphabet in order. This is often given as an example for the Playfair cipher.

The original version was more complicated starting with a matrix the same width as the keyword:

```
INCTAUS
BDEFGHK
LMOPQRV
WXYZ
```

Then rewriting the columns into *rows* of a 5×5 matrix:

```
IBLWN
DMXCE
OYTFP
XAGQU
HRSKV
```

The message in broken into letter pairs and these pairs are replaced using the matrix with the following rules:

- If a pair of letters lie in the same row they are replaced by the letters on their right. (`EC` becomes `DE`)
- If they lie in the same column they're replaced by the letter below (`GL` ⇒ `SX`)
- If they're not in the same row or column the first cipher letter is found in the intersection of the row of the 1st letter with the column of the second, and the 2nd cipher letter is at the inersection of the row of the 2nd plaintext letter with the column of the first. (`HE` ⇒ `VD`)

(See Brian Beckett's *Introduction to Cryptology*, page 169.)

Vigenère's Autokey Ciphers

These ciphers, in which the message acts as its own key, were developed by Vigenère and are more complex than the cipher he is usually credited with.

The idea is to use a seed character to start the key, and as with an ordinary Vigenère cipher the first plaintext character is replaced by the character n places along in the alphabet, where n is determined by the seed character. If the seed is `d`, the shift is 4 because `d` is the 4th letter in the alphabet and the plaintext character `i`, for instance, would be replaced by `m` in the ciphertext. Instead of using a second key character as with the Vigenère cipher to encrypt the second plaintext letter, the first plaintext character in use as the key for the second letter and so on for the rest of the plaintext.

In short, the first letter is enciphered using the seed, and each successive plaintext letter is enciphered using the plaintext letter that came before it. Here is an example with the seed letter `j`:

```
 plaintext: l  o  n  d  o  n  c  a  l  l  i  n  g  m  o  s  c  o  w
  key text: j  l  o  n  d  o  n  c  a  l  l  i  n  g  m  o  s  c  o
     shift: 10 12 15 14  4 15 14  3  1 12 12  9 14  7 13 15 19  3 15
ciphertext: v  a  c  r  s  c  q  d  m  x  u  w  u  t  b  h  v  r  l
```

All that is needed to decode the text is the seed letter. Shifting the first ciphertext letter back the number of places in the alphabet corresponding to the seed reveals the first plaintext letter, which is then used to decipher the second ciphertext letter and so on.

One Time Pad (Vernam cipher)

The One Time Pad (OTP) or Vernam cipher is a mathematically unbreakable cipher although it is not commercially practical. It simply exclusive-or's (XOR, denoted by $\oplus$) the characters of the plaintext with the characters of a randomly generated key of the same length as the plaintext.

Conventionally the plaintext is represented in binary form. The usual means is to represent each letter of the alphabet by a number and to write the number as a binary (base 2) number with a fixed number of digits. The letters in the alphabet can be represented by the numbers between 0 and 25 which can all be written as 5 digit binary numbers, e.g.:

```
  A     B     C     D     E
00000 00001 00010 00011 00100

  F     G     H     I     J
00101 00110 00111 01000 01001
```

and so on up to `z` which is represented by `11001`.

The next stage is to generate a key which is a random series of 1's and 0's with the same number of digits as the message. The message (plaintext) is then XORed with the key to produce the ciphertext. XOR is a logic operation applied to the nth digit of the plaintext and the nth digit of the key to produce the nth digit of the cipher text. The XOR rule is laid out in the following table:

$P \oplus K$	$\Rightarrow$	C
$0 \oplus 0$	$=$	0
$0 \oplus 1$	$=$	1
$1 \oplus 0$	$=$	1
$1 \oplus 1$	$=$	0

where

P	$=$	plaintext digit
K	$=$	key
C	$=$	ciphertext digit

i.e. if the two P and K digits are the same they XOR to give 0 and if they are different they XOR to give 1.

Provided the key generation is truly random all possible ciphertexts are equally likely. A corollary of this is that for a given ciphertext all possible plaintexts are equally likely and thus if you don't know the key the code is unbreakable.

```
          L     O     N     D     O     N     C     A     L     L     I     N     G
Text:   01011 01110 01101 00011 01110 01101 00010 00000 01011 01011 01000 01101 00110
Key:    01001 10010 11101 01110 00100 01010 00110 01001 10110 11101 00010 01011 01001
Cipher: 00010 11100 10000 01101 01010 00111 00100 01001 11101 10110 01010 00110 01111

Cipher string: 0001011100100000110101010001110010001001111011011001010001100111 1
```

To decode, simply XOR the key with the ciphertext to produce the plaintext (examining the above example will show that this works).

Random numbers can be generated using a random number generator program, however such a program only produces pseudo random numbers and thus key generated this way are ultimately vulnerable. To produce a *truly* random key it is better to use a quantum machanical event such as nuclear decay to generate your 1's and 0's - you could use the clicks on a geiger counter - assigning a 1 for a long gap between clicks and a 0 for a short gap (choosing the threshold time length between long and short to give an equal number of 1's and 0's). This would result in a truly random key and thus an unbreakable cipher - provided a given key is only used once.

Weaknesses

If you use the same key twice on two different messages you render the messages much more vulnerable to attack. Your enemy can XOR the two ciphertexts against one another to produce a text which is the same as that obtained by XORing the two original plaintext messages. This is the same as having a copy of either message encrypted by a non-random key and reduces the problem to that of breaking the Augustus cipher. The math behind this is illustrated below:

$$\begin{aligned} P1 &= \text{plaintext 1} \\ P2 &= \text{plaintext 2} \\ C1 &= \text{ciphertext 1} \\ C2 &= \text{ciphertext 2} \\ K &= \text{key} \\ \oplus &= \text{XOR operation} \end{aligned}$$

Exclusive-or is associative and commutative, therefore:

$$\begin{aligned} A \oplus B &= B \oplus A \\ (A \oplus B) \oplus C &= A \oplus (B \oplus C) \\ &= A \oplus B \oplus C \\ A \oplus A &= 0 \\ A \oplus 0 &= A \end{aligned}$$

The two ciphertexts are produced using the same key:

$$\begin{aligned} C1 &= P1 \oplus K \\ C2 &= P2 \oplus K \end{aligned}$$

Your enemy intercepts the two ciphertexts and does the following:

$$C1 \oplus C2 = (P1 \oplus K) \oplus (P2 \oplus K)$$

because XOR is associative,

$$= P1 \oplus K \oplus P2 \oplus K$$

because XOR is commutative,

$$= P1 \oplus P2 \oplus (K \oplus K)$$

because A $\oplus$ A = 0,

$$= P1 \oplus P2 \oplus 0$$

because A $\oplus$ 0 = A,

$$= P1 \oplus P2$$

Since neither $P1$ nor $P2$ is random, your enemy now has a much simpler problem to solve, and if they succeed they will not only decode the messages $P1$ and $P2$, but will also have the key, K, since $P1 \oplus C1 = K$.

Hence this cipher is called the One Time Pad for a very good reason.

WHAT THE OTHER GUY IS DOING

LEGRAND (Charles Keane) is learning MacPearl and is concentrating on solving P's. He is looking for members of the Krewe that are experts at solving P's who would like to correspond.

GUNNER (Paul Brothers), is looking for suggestions and/or recommendations on learning and improving his proficiency in Power Basic. He's using a 486 with DOS 6.2, Windows, and OS/2.

Robert Rutherford is looking for some details on **BITSIFTER's** `CRYPTO`. He writes, "The documentation for the maintenance program at step 6 shows `FINAL` and at step 7 shows `ERASE *.JNK`. Is `FINAL` another program that is hidden ?" Any members of the Krewe that are familiar with `CRYPTO`, please write or e-mail the Editor.

PLEA FOR ARTICLES

The in-bins are empty! The *Computer Supplement* is a compilation of articles and programs sent in by subscribers. Right now, there are no further articles waiting to be published.

If you're working on a cipher, write up a short description and send it in. If you've come up with a great piece of software that helps your solving, send it in!

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.

THE MATHEMATICAL GUTS OF RSA ENCRYPTION

Francis Litterio

Here's the relatively easy to understand math behind RSA public key encryption. It is most ironic that it is *not* illegal to transport this mathematical description out of the U.S. and Canada, but it *is* illegal to export an implementation of this cipher from the U.S. or Canada. Any competent programmer could use this knowledge to implement a strong crypto product beyond the borders of the U.S. and Canada.

Directions

1. Find P and Q, two large (e.g., 1024-bit) prime numbers.
2. Choose E such that E and $(P-1)(Q-1)$ are *relatively prime*, which means they have no prime factors in common. E does not have to be prime, but it must be odd. $(P-1)(Q-1)$ can't be prime because it's an even number.
3. Compute D such that $(DE-1)$ is evenly divisible by $(P-1)(Q-1)$. Mathematicians write this as

   $DE = 1 \bmod (P-1)(Q-1)$

   and they call D the *multiplicative inverse* of E.

Definitions

- The encryption function is

  $encrypt(T) = (T^E) \bmod PQ$

  where T is the plaintext (a positive integer).
- The decryption function is

  $decrypt(C) = (C^D) \bmod PQ$

  where C is the ciphertext (a positive integer).
- Your *public key* is the pair (PQ, E).
- Your *private key* is the number D (reveal it to no one).
- The product PQ is the *modulus*.
- E is the *public exponent*.
- D is the *secret exponent*.

Notes

You can publish your public key freely, because there are no known easy methods of calculating D, P, or Q given only (PQ, E) (your public key). If P and Q are each 1024 bits long, the sun will burn out before the most powerful computers presently in existence can factor your modulus into P and Q, using presently known factoring algorithms.

Caveats

Given all the hype that RSA is getting these days, it's worthwhile to keep the following caveats in mind.

- It is not yet rigorously proven that no easy methods of factoring exist. Improvements to factoring algorithms are being made, as well as speed improvements in computer hardware.
- It is not yet rigorously proven that the only way to crack RSA is to factor the modulus.