4-1985

Rivest-Shamir-Adelman Cryptosystem

Edward A. Fetzner
Longwood University

Rivest-Shamir-Adelman
Cryptosystem

By
Edward A. Fetzner
April 1985

Director: Dr. Robert Webber

Robert P. Webber

Reader 1: Dr. Robb Koether

Robb T. Koether

Reader 2: Dr. Robert May

Robert M. May

Reader 3: Dr. E.T. Noone

E. T. Noone Jr

Committee Chair: Dr. Rosemary Sprague

Rosemary Sprague

Rivest-Shamir-Adelman Cryptosystem

By
Edward A. Fetzner
April 1985

A paper presented to Longwood College in partial fulfillment of the requirements for honors in Computer Science

INTRODUCTION

Definition:

Cryptography, the art of writing secret messages, has been practiced for almost 3000 years. Two people have a message to communicate to each other, with a chance of an outside person intercepting the message before it reaches the receiver. Therefore, the message must be put through an encryption scheme to keep the intervening person from finding the true nature of the message. An encryption scheme is coding a message using an enciphering key and then decoding the coded message with a deciphering key.

Types of Encryption Schemes:

The earliest encryption schemes are known as symmetric schemes, since the enciphering and deciphering keys are identical or they are related in such a way that by knowing one, the cryptographer can derive the other from it. The best example of this type of scheme is the Caesar Cipher [5, pp. 69-71], which consists of first converting the characters of the message into numbers.

Example 1: Ceasar Cipher

Let A=01
B=02
C=03
:
:
Z=26

Using EUCLID as the message, the number blocks would look like:

E	U	C	L	I	D
05	20	03	12	09	04

Next, a number is chosen that is larger than the number of symbols in the alphabet used. Then, the number chosen is added to each number block. Next, the number of symbols in the alphabet used is divided into the sum just mentioned and the remainder, or modulus, is the coded block. The last step in the coding process is to take the coded blocks and convert them back to characters.

Example 2: Caesar Cipher continued

If the ciphering key is 31 and the English alphabet is is used (26) then the coded blocks would be:

	10	25	08	17	14	09
or	J	Y	H	Q	N	I

To decode the coded message, subtract each coded block from the the number chosen, take the modulus, and convert back to characters.

With today's computers, symmetric encryption schemes are virtually useless, since a computer could do all the calculations to decode the message in seconds. Also, computers are handling more and more of our communications, and since these communications are vulnerable to interception, encryption schemes have to be designed to be computer unbreakable. These encryption schemes are known as asymmetric, since the ciphering and deciphering keys are different, and it is computationally infeasible to derive one from the other.

THE RSA

Definition:

The Rivest-Shamir-Adelman [6], or RSA, cryptosystem currently is computer unbreakable. The RSA is an asymmetric encryption that incorporates a one-way algorithm. A one-way algorithm is invertible,

and it is easy to compute all values needed for the ciphering and deciphering keys. But for all domain values of one key it is computationally infeasible to compute the value of the inverse of that key, even if a complete description of the algorithm is given.

Values Needed to Find the Ciphering and Deciphering Keys:

The values needed for the cipher and decipher keys are two prime numbers. A prime number is a positive integer that can be divided, with a remainder of 0, only by one and the number itself. Next, the product of the two primes is found and the Euler phi (see APPENDIX 1 for a definition of Euler phi value) value of the product of the two primes is calculated. The Euler phi value is found by subtracting one from each prime and then multiplying them together. This gives the following information:

Two Primes: P & Q
N = P X Q
O(N) = (P-1) X (Q-1) (Euler Phi value)

Finding the Ciphering and Deciphering Keys:

Now the ciphering key, C, can be derived by finding a number that has a greatest common divisor of one with O(N), or (C,O(N))=1. A greatest common divisor (GCD) is the largest number that will divide into two numbers with a remainder of zero. For example, the GCD of 6 and 9 is 3, or (6,9)=3.

Next the deciphering key, D, is derived by finding a number that is a multiplicative inverse of C modulo O(N). This is a number that, if multiplied with C and divided by O(N), would have a remainder of one. The best way of finding the GCD and the multiplicative inverse of C mod O(N) is to use the Euclidean Algorithm [1, pp. 28-30]. The

Euclidean Algorithm is a series of repeated subtractions that will yield a 1 as a last nonzero remainder, if C and O(N) have a GCD of one. Then, if the steps are reversed, using substitution, the deciphering key will be the number multiplied with C after the last substitution.

Example 3: Euclidean Algorithm

If C=167, P=47, Q=61 then: N=47X61=2867
O(N)=46X60=2760

First check to see if (C,O(N))=1:

1A) 2760=(167X16)+88 remainder
2A) 167=(88X1)+79
3A) 88=(79X1)+9
4A) 79=(9X8)+7
5A) 9=(7X1)+2
6A) 7=(2X3)+1 Last non-zero remainder. Therefore the
7A) 2=(1X2)+0 GCD of 167 and 2760 is 1.

Now reverse the steps and substitute to find D:

1B) 1=7-(2X3) Substituting 5A for 2 in 1B
2=9-(7X1)

2B) 1=7-(9-(7X1))X3

3B) 1=7X4-9X3 Substituting 4A for 7 in 3B
7=79-(9X8)

4B) 1=(79-(9X8))X4-9X3

5B) 1=79X4-9X35 Substituting 3A for 9 in 5B
11=53-(14X3)

6B) 1=79X4-(88-(79X1)X35

7B) 1=79X39-88X35 Substituting 2A for 79 in 7B
79=167-(88X1)

8B) 1=(167-(88X1))X39-88X35

9B) 1=167X39-88X74 Substituting 1A for 88 in 9B
88=2760-(167X16)

10B) 1=167X39-(2760-(167X16))X74

11B) 1=167X1223-2760X74 Last substitution

Therefore, 1223 is the multiplicative inverse of 167 modulo 2760.

RSA Algorithm:

Now that all the values have been calculated, the RSA algorithm can be easily explained. The first step is to convert the characters

of the message into numbers (as in example 1), then block them into groups.

Example 4: RSA NUMBER BLOCKS

Using example 1 and blocking the characters by 2, the number blocks would look like:

```
 E U    C L    I D
0520   0312   0904
```

** Note that the number blocks can be grouped by any number of characters.

Then each number block is raised to the power of the ciphering key and divided by N to produce the remainder, which is the coded block. To decode the coded blocks, raise each block to the deciphering key, divide by N to get the remainder, and convert back to characters. Letting NB be the uncoded blocks, CB be the coded blocks, and ** mean raised to a power, the algorithm would look like:

CB=NB**C MOD N

NB=CB**D MOD N

The MOD in the previous equation means to find the remainder of NB**C divided by N.

Example 5: RSA ALGORITHM

Using the the number blocks in example 4 and the calculated value in example 3, the RSA algorithm works as follows:

To code the message:

CB1=520**167 MOD 2867=1058
CB2=312**167 MOD 2867=315
CB3=904**167 MOD 2867=621

The coded message: 1058, 315, 621

To decode the message:

NM1=1058**1223 MOD 2867=520

NM2=315**1223 MOD 2867=312
NM3=621**1223 MOD 2867=904

Convert the number blocks back to characters.

PROBLEMS IN PROGRAMMING AN RSA CRYPTOSYSTEM

Large numbers:

Even though the algorithm is easy to compute with relatively small numbers, to make the algorithm truly computer-unbreakable the two primes chosen should be at least 100 digits long, and the ciphering and deciphering keys should be over 5 digits. Working with numbers of this size is very slow even for computers. For example, to raise 9,999 to the 9,999 power using multiplication might take a computer around 30 seconds. But if the number blocks are grouped by twos, then 9,999 represents two characters and in a passage of only 100 words (approximately 400 characters), those few seconds could become hours. Therefore a more efficient way should be used.

Another problem with computers and large numbers is that computers usually divide by repeated subtraction. If the number to be divided is 100 times larger then the number dividing it, a computer will take a few seconds to produce an answer. As before, with a 100 word passage the waiting time could be in the hours.

Solutions:

In order to speed up both the exponentiation process and keeping a large number from being divided by a small number, a Russian Peasant Algorithm [2, pp. 43-45] can be used. The Russian Peasant Algorithm first converts the exponent to a binary number.

Example 6: BINARY NUMBER

167 in base 2 is: 10100111.

The following table shows what each position in the binary number represents. Row one is the power of 2 for that position, row two is the binary number, and row three is the base ten representation.

(row 1)	2**7		2**6		2**5		2**4		2**3		2**2		2**1		2**0
(row 2)	1		0		1		0		0		1		1		1
(row 3)	128	+	0	+	32	+	0	+	0	+	4	+	2	+	1=167

Secondly, the algorithm uses the following rules to raise the base number to the required exponent and find the modulus.

1) Read the binary number, giving the exponent, left to right.
2) Start with 1 as preliminary "result".
3) A 0 digit means square "result" to get new "result".
4) A 1 digit means square "result" and multiply with base, or original number, to get new "result".
5) After each squaring and multiplication with base, take the modulus with respect to N, to get the new "result".

Example 7: RUSSIAN PEASANT ALGORITHM

To compute CB1 in example 5:

Binary number	Calulations	
1	(1**2)X520=520	(Rule 2)
0	520**2 MOD 2867=902	(Rule 3 & 5)
1	902**2 MOD 2867=2243	(Rule 4 $ 5)
	2243X520 MOD 2867=2358	(Rule 4 & 5)
0	2358**2 MOD 2867=1051	(Rule 3 & 5)
0	1051**2 MOD 2867=806	(RULE 3 & 5)
1	806**2 MOD 2867=1694	(Rule 4 & 5)
	1694X520 MOD 2867=711	(Rule 4 & 5)
1	711**2 MOD 2867=929	(Rule 4 & 5)
	929X520 MOD 2867=1424	(Rule 4 & 5)
1	1424**2 MOD 2867=807	(Rule 4 & 5)
	807X520 MOD 2867=1058	(Rule 4 & 5)

So instead of taking 167 multiplications and then taking the modulus of that large number, the Russian Peasant Algorithm only takes 12 steps, and the result of each squaring is kept close to the size of the modulus.

Also, long division can be used to speed up the run time for calculating the remainder. Long division subtracts the divisor from the left of the given number. Each time the left of the number is reduced to smaller than the divisor, the next number to the right is added and the divisor is subtracted again. This is continued until the right most number is used and the number is smaller than the divisor.

Example 8: LONG DIVISION

Find the remainder of 79435 when dividing by 715 using long division:

```
 79435
-715          (subtract from the left)
  79          (since the left of the number is
               smaller than the divisor, bring
 793           the next number on the right.)
-715          (subtract)
  78          (number is smaller than divisor)

 785          (bring down next number)
-715          (subtract)
  70          (remainder)
```

Lists:

Another very important idea to remember when working with computers and very large numbers is memory efficiency. Since a number over 9 digits is too large to keep in integer form, the number needs to be broken into a list of digits.

Example 9: LIST

The number 8,469,843,132, would look like this in a list:

number:	8	4	6	9	8	4	3	1	3	2
position in list	1	2	3	4	5	6	7	8	9	10

There are two ways in which a list can be built and maintained: arrays and linked lists. Arrays are easier to work with since any part of the list can be accessed, but the size of the array must be preset. For example, in the computer language FORTRAN [4, pp. 64-70], a linked list is an array of preset size, with several fields in each position. The extra fields contain the position of the next or previous position in the list.

Example 10: LINKED LIST ARRAY

Using the number in example 9, it would look like this in an array linked list:

Fields	Data									
position:	1	2	3	4	5	6	7	8	9	10
number:	8	4	6	9	8	4	3	1	3	2
forward pointer:	2	3	4	5	6	7	8	9	10	nil
back pointer:	nil	1	2	3	4	5	6	7	8	9

The nil in the first and tenth positions designates the end of the list. For instance, the tenth position in the forward pointer row is nil since there is no eleventh position to "point" to.

Presetting the size of the array can waste space. For example, if zip codes are to be put into a list, the preset size would be 9 positions, since some locations now use 9 digit instead of 5 digit zip codes. Therefore, every time a 5 digit zip code was put into a list there would be four unusable positions in memeory The major drawback with using arrays with the RSA is that the array would have to be set

to the maximum length of the modulus. But the remainder, if working with 100 digit primes, could be between 1 and 200 digits. This could make up to 199 positions of memory unusable. A programmed RSA cryptosystem uses up to 10 of these arrays, which could render 1990 memory positions unusable.

Solution:

The program needs to be able to create link lists without presetting the size of the list. The best language for accomplishing this is Pascal [3, pp. 395-410]. In Pascal, only the "type" of record is preset. For instance, if each cell of the list is to contain a block of data and two pointers, then at the beginning of the program a record type is set up to have one block of data and two pointers. From that point on, to add to the list a new block is created and the front and back pointers are set. If the list shortens then the extra blocks can be disposed of easily.

FINAL NOTE

Most of the problems in programming an RSA cryptosystem is working with large numbers, specifically the large primes. However, these large primes are what makes the RSA computer unbreakable. To break the RSA, the product of the two primes would have to be factored in order to obtain the primes themselves, and the decipher number would have to also be found to decode the message. For example, if two 100 digit primes were used, then the product of these two would be a number around 200 digits long. If a trial and error process is used to find the two primes, it could take trillions of trials to factor the product. Also, to find a deciphering key, (that is, an inverse of some

number modulo the Euler phi value of the two 100 digit primes), could take another trillion or so trials. The amount of time, even using the most advanced computer system, to break the RSA would take thousands of years. Therefore, until a more effective way of factoring large numbers is discovered, the RSA remains computer unbreakable.

HINTS FOR PROGRAMMING THE RSA

The following miscellaneous items are very helpful to remember in programming a RSA cryptosystem:

1) Use structured progaming techniques.

2) The number blocks and coded blocks must be smaller than the modulus.

3) The Pascal standard functions, ORD & CHR, are very good ways to convert the characters into numbers and numbers back to characters respectively.

4) Make sure that the multiplication and division routines take into account all possible situations that may occur when multiplying and dividing large numbers.

5) Make sure that all blocks created, if using Pascal, are disposed of properly or memory may be exhausted.

APPENDIX 1

DEFINITION: If N>1, let O(N) denote the number of positive integers which are less than N and relatively prime to N.

**Note: The reason for using the Euler phi value is to remove the restriction of N being a prime.

Bibligraphy

1 Armendaiz, Efraim, & McAdam, Stephen, Elementary Number Theory, Macmillan Publishing Co., Inc., New York, 1983.

2 Baskst, Aaron, Mathematical Puzzles and Pastimes, D Van Nostrand Company, Inc., Princeton, New Jersey, 1954.

3 Cooper, Doug, & Clancy, Michal, Oh! Pascal, W. W. Norton & Company, Inc., New York, N. Y., 1982.

4 Day, Colin, Fortran Techniques, Cambridge University Press, 1972.

5 Konheim, Alan, Cryptography a Primer, John Wiley & Sons, Inc., New York, 1981.

6 Rivest, R. L., Shamir, A., & Adelman, L., "A Method For Obtaining Digital Signatures and Public-Key Cryptosystems," MIT Lab. Comp. Sci. Rep., MIT/LCS/TM82 (technical memo), 1977.

PSUEDO CODE
OF THE RSA

This is the pseudo code for the Rivest-Shamir-Adelman cryptosystem. The program will take the inputted file and code or decode it, depending on the user's needs. The program first checks to see if the user has a valid code number to use the system. This is done by checking the users inputted code number with the file of code numbers. If the code number is bogus then the user is disconnected from the system. Otherwise all the values needed for the algorithm are read in and the coding or decoding process is continued until the end of the inputted file is reached.

The RSA cryptosystem is broken into the following procedures:

Link list procedures:
- INTEGER_TO_LINK_LIST
- CREATE_DUPLICATE_LIST
- LIST_TO_DBL_LIST

Reading in the data file procedures:
- READCIPHTEXT
- READDECIPHTEXT

Arithmetic procedures:
- BINARYCON
- MULT_NUMBERS
  - ADD_NUMS (Subprocedure of MULT_NUMBERS)
- MODULUS
  - UPDATE (Subprocedure)

Introduction to cryptosystem:
- INTRO
  - CHECKNUM
  - SETCODES

Writing procedures:
- WRITE_CIPHER_TEXT
- WRITE_DECIPH_TEXT

(**Note: Check APPENDIX 1 for a complete variable listing and discreption.)

Link list procedures

Procedure INTEGER_TO_LINK_LIST will take the integer from READCIPHTEXT and put it into the link lists, BASE & RESULT. The integer is put in a link list since its value will become to large too keep in integer form. The digits of the number are stored backward in the list with a -1 as a nil pointer.

```
Begin
  Raise PUTVAR2 to the approprate power determined by WORDSIZE.
  Extract the last digit from given integer, LISTNUM.
  If the last digit is a 0 then
```

```
   While the last digit is 0 do
    Begin
     Delete last digit from number
     Reduce PUTVAR2 by 10
     Extract next-last digit from LISTNUM
    End
  While LISTNUM is greater or equal to 10 do
   Begin
    Create new block in list RESULT
    Store digit from LISTNUM in block
    Set pointer of new block to point to old block
    Reset head of the list to new block
    Repeat the previous four steps with list BASE
    Delete digit from LISTNUM
    Reduce PUTVAR2 by 10
    Extract next digit from LISTNUM
   End of while LISTNUM is greater or equal to 10
  Create new block in list RESULT
  Store first digit of LISTNUM in block
  Set pointers of old block to new block
  Set head of the list to new block
  Repeat previous four step with list BASE
End of procedure INTEGER_TO_LINK_LIST.
```

Procedure CREATE_DUPLICATE_LIST will take a given link list and dulpicate it using the second given link list. This is used for squaring a number.

```
  Begin
   Set first given link list, CDLNUM1, to start of list
   Set DUMPOINTER to first block in second link list, CDLNUM2
   While CDLNUM1 is not equal to the end of the list do
    Begin
     Store CDLNUM1 block into CDLNUM2 block
     Creat new CDLNUM2 block
     Set old block (DUMPOINTER) to point to new block
     Advance CDLNUM1 list to next block
    End of while CDLNUM1 is not equal to the end of the list
   Set end of the list value for CDLNUM2
  End of procedure CREATE_DUPLICATE_LIST
```

Procedure LIST_TO_DBL_LIST will put a given singly linked list into a doubly linked list. A doubly link list is needed by the MODULUS procedure for division.

```
  Begin
   Set the doubly link list, DBLNUM2, to start
   Set the number head of list
   Set DUMDBL pointer to current position of DBLNUM2
   Create new block in DBLNUM2
   Set the pointer of new block to point to the old block, DUMDBL
   Set the pointer of old block to point to the new block
   Set DUMDBL to the new block
   Set the head of the list to current block in DBLNUM2
```

```
Set the singly link list DBLNUM1 to start
While DBLNUM1 is not at the end of the list do
 Begin
  Store DBLNUM1 in current block of DBLNUM2
  Create new block in DBLNUM2
  Set DUMDBL pointer to point to new block
  Set the current block to point to old block, DUMDBL
  Set DUMDBL to new block
  Advance DBLNUM1 to next block in list
 End of while DBLNUM1 is not at the end of the list
Set the tail pointer to the current block in DBLNUM2
Store the number end of the list for DBLNUM2
Create new block in DBLNUM2
Set DUMDBL to point to new block
Store the number end of the list for DBLNUM2
Set pointer of current block in DBLNUM2 to point to DUMDBL
End of the procedure CREATE_DUPLICATE_LIST.
```

Reading in the data file procedures:

Procedure READCIPHTEXT will read from the data file the number of characters according to the constant, SIZE-1, one at a time. The characters will be turned into number values by ORD, raised to the appropriate powers and added together to produce a single number which will then be put into a link list.

```
Begin
 Create new block in list RESULT
 Store a -1 for the nil pointer at the tail of the list
 Set the head pointer to the current head of the list
 Create new block in list BASE
 Store a -1 for the nil pointer at the tail of the list
 Set the head pointer to the current head of the list
 Set end of file flag, MAINFLAG, to off position
 If the end of the file has not been reached then
  Begin
   Read first character from text OLDTEXT
   If the end of the line marker is found then
    Begin
     Reset file OLDTEXT to next line in file
     If the end of the file has not been found then
      Read next character from OLDTEXT
     Else if the end of the file was found after resetting then
      Begin
       Set flag, READFLAG, to end the procedure
       Set flag, MAINFLAG, to end the program
      End of else statement
     End of resetting file to next line
    If the end of the procedure flag is off then
     Begin
      Change the character read into a number and store in REPROD
      If REPROD is larger then 99 then
       Subtract 99 to keep all character numbers to two digits
      Multiply REPROD by 100 to be able to combine with next
```

```
    character read.
   If the end of the file has not been found then
    Begin
     Read next character from OLDTEXT
     If the character read is the end of the line marker then
      Begin
       Reset file to next line
       If the end of the file is not found then
        Read the next character from the file
       Else if the end of the file is found then
        Begin
         Set flag to end the procedure
         Set flag to end the program
        End of the Else statement
       End of resetting the file statement
      If there was a second character read then
       Begin
        Change the second character read to a number
         and store in RESUM
        If RESUM is greater then 99 then
         subtract 99 to keep all charcter number to two digits
        Add RESUM and REPROD, to form the number block
       End of if there was a second character
      End if the end of the file has not been found
     Else if there was no second character then the number block
      is just the first character times 100
     Call the procedure INTEGER_TO_LINK_LIST to put the number
      block into a link list
    End of if the end of the procedure flag is off
   End of if the end of the file was not found at the beginning
  End of the procedure EADCIPHTEXT.
```

Procedure READDECIPHTEXT will read a coded number from the file OLDTEXT until it reads a ".", which is the end of the coded number marker. A check for the end of the line is also made to reset the file to the next line. Once the end of the file is found the flag MAINFLAG will be set to the on position to end the program.

```
Begin
  Create a new block in list RESULT
  Set the head of the list
  Set a dummy pointer to the current block in list
  Read the first number to be stored into REDCHAR
  While REDCHAR is not equal to "." do
   Begin
    Since the numbers are read as characters, change the character
     number to an integer (see APPENDIX II)
    If the character read is a blank then
     set the end of the file flag, MAINFLAG, to the on position
    Store the read number in the list
    Create new block in list
    Set pointer of the old block to point to new block
    Set the dummy pointer to new block
    If the end of the line has not been reached then
```

```
    Read the next digit in number
   Else if the end of the line was been reached then
    Begin
     Reset the file to the next line in the file
     Read the next digit in number
    End of else if the end of the line was reached
   End of while not the end of the coded number
  Set the end of the list marker
  Create new block in list BASE
  Set the head of the list for BASE
  Call procedure CREATE_DUPLICATE_LIST to create list BASE using
   list RESULT
End of the procedure READDECIPHTEXT.
```

ARITHMETIC PROCEDURES

Procedure BINARYCON will take a given integer, the ciphering or deciphering key, and change it to a binary number. This is used to speed up raising the number block to be coded or decoded to the given power.

```
Begin
  Set BINPOW to equal the given power
  Set the binary array position counter to the start of the array
   While BINPOW is greater or equal to 2 do
   Begin
    If BINPOW is an odd number then
     Store a 1 in the current position in the array
    Else if BINPOW is a even number then
     Store a 0 in the current position in the array
    Advance the array position counter to the next position in array
    Divide BINPOW by 2 to get the next number for the array
   End of while BINPOW is greater or equal to 2
  If BINPOW is reduced to zero then
   Store additional 0 and then a 1 in the array
  Else if BINPOW is not reduced to 0 then
   Store an additional 1 in the array
End of the procedure BINARYCON.
```

Procedure MULT NUMBERS will take two link lists and multiply them together. This is done by taking each digit in link list, NUMBER2, one at a time, and multpling it with every digit in link list NUMBER1. The first product is stored in the link list MULTNUM1 and all of the rest of the products are stored in MULTNUM2. After the first product MULTNUM1 will be used as a running total of the products. Also after the second product, MULTNUM1 and MULTNUM2 are added together after each product is produced.

```
Begin
  Create a new block in MULTNUM1
  Set the head of the list to that block
  Create a new block in MULTNUM2
  Set the head of the list to that block
  Set NUMBER1 to the start of the list
```

```
Set NUMBER2 to the start of the list
Set the number of zeros to be embedded to starting number
Set a dummy pointer to the current block in MULTNUM1
Set a second dummy pointer to the current block in MULTNUM2
Set the flag to indicate a carry, MULTFLAG1, to off
Set the flag for embedding zeros, MULTFAG2, to on
Set the pass counter, MULTCOUNT, to first pass
Set the flag to indicate a carry on the last multiplication,
 MCFLAG, to off
While the pass counter, MULTCOUNT, is not equal to 0 do
 Begin
  While the end of the list, NUMBER1, is not reached do
   Begin
    If there is no carry from the previous multiplication then
     Multiply the current block of NUMBER1 and NUMBER2 together
    Else if there is a carry from the previous multiplication then
     Multiply NUMBER1 and NUMBERS together and add the carry
    If the product is zero then store a 0 in MULTMOD
    Else MULTMOD equals the product modulo 10
    Set the carry flag, MULTFLAG1, to the product divided by 10
    If it is the first pass then
     Begin
      Store MULTMOD in the list MULTNUM1
      Create a new block in MULTNUM1
      Set the old block to point to the new block
      Store the nil pointer of the list
      If the last carry flag, MCFLAG, is on then
       Reset the nil pointer in NUMBER1
       Else if MCFLAG is off then
       Advance NUMBER1 to the next number in the list
     End of if it is the first pass
    Else if it is not the first pass then
     Begin
      If the flag to embed zeros, MULTFLAG2, is on then
       For 1 to the number of zeros needed, MULTVAR2, do
        Begin
         Store a 0 in MULTNUM2
         Create a new block in MULTNUM2
         Set the pointer of the old block to point to the new block
         Set the dummy pointer to the new block
         Turn MULTFLAG2 off
        End of embedding zeros
       Store MULTMOD in list MULTNUM2
       Create a new block in MULTNUM2
       Set the pointer of the old block to point to the new block
       Store nil pointer of the list
       If the last carry flag, MCFLAG, is on then
        Reset the nil pointer in NUMBER1
       Else advance NUMBER1 to the next number in the list
      End of if it is not the first pass
     If the end of the NUMBER1 is found but there is a carry from
      the previous multiplication then remove the nil
      pointer in NUMBER1 to enable the routine to make one more
      pass to store the carry
```

```
    End of while not the end of NUMBER1
   If it is not the first pass then
    Call procedure ADD_NUMS to add MULTNUM1 and MULTNUM2
   If it is not the first pass then
    Begin
     Dispose of the list MULTNUM2
     Create new block in list MULTNUM2
     Set the head of the list
    End of disposing of MULTNUM2
   Add one to the number of zeros to be embedded, MULTVAR2
   Add one to the pass counter, MULTCOUNT
   Advance NUMBER2 to the next number in the list
   If the next number in NUMBER2 is a zero then
    Until a non-zero number is reached do
     Begin
      Advance NUMBER2 to the next block
      Add one to MULTVAR2
     End of until a non-zero number is reached
   If the end of NUMBER2 is reached then
    Set the pass counter, MULTCOUNT, to zero
   Set a dummy pointer to the current block in MULTNUM2
   Set the embed zero flag, MULTFLAG2, to on
   Set the last carry flag, MCFLAG, to off
  End of while pass counter is not equal to zero
 Set NUMBER1 and MULTNUM1 to start of the list
 While the end of MULTNUM1 is not reached do
  Begin
   If the end of NUMBER1 is not reached then
    Begin
     Store the current block of MULTNUM1 in NUMBER1
     Advance both list to the next block
    End of if the end of NUMBER1 is not reached
   Else if the end of NUMBER1 is reached then
    Begin
     Set a dummy pointer to current block in NUMBER1
     Store MULTNUM1 in NUMBER1
     Create a new block in NUMBER1
     Set the pointer of the old block to point to the new block
     Set the nil pointer in NUMBER1
     Set a dummy pointer to the current block in NUMBER1
    End of else if the end of NUMBER1 is reached
   End of while not the end of MULTNUM1
  Dispose of MULTNUM1
  Dispose of MULTNUM2
 End of the procedure MULT_NUMBERS.
```

Subprocedure ADD_NUMS is a procedure that is part of MULT_NUMBERS. This procedure will add the running total of the products, link list MULTNUM1, and the current multiplication in MULTNUM2. Corresponding blocks in both lists are added and stored in the current block of MULTNUM1.

```
Begin
  Set MULTNUM1 AND MULTNUM2 to the beginning of the list
```

```
Set the carry flag, ADDFLAG, to off
While MULTNUM1 & MULTNUM2 are not at the end of the list do
 Begin
  If the carry flag, ADDFLAG, is off then
   Add MULTNUM1 & MULTNUM2 and store in ADDSUM
  Else if ADDFLAG is on then
   Begin
    Add MULTNUM1, MULTNUM2, and add one to that sum
    Store in ADDSUM
    Turn ADDFLAG off
   End of else if ADDFLAG is on
  Subtract 10 from ADDSUM
  If ADDSUM is negative then
   Begin
    Add 10 to ADDSUM
    Store ADDSUM in current block in MULTNUM1
   End of if ADDSUM is negative
  Else if ADDSUM is positive then
   Begin
    Store ADDSUM in MULTNUM1
    Turn ADDFLAG on
   End of else if ADDSUM is positive
  Advance MULTNUM1 & MULTNUM2 to next block in list
 End of while MULTNUM1 & MULTNUM2 are not at the end of the list
Set a dummy pointer to current block in MULTNUM1
If there is a carry from the last addition and the end of the list
 MULTNUM2 has been reached then
 Begin
  Store a one in the last block of MULTNUM1
  Create a new block in MULTNUM1
  Set the pointer of the old block to point to the new block
  Store a nil pointer in new block
 End of if there is a last carry
Else if the end of MULTNUM2 has not been reached then
 While MULTNUM2 is not at the end of the list do
  Begin
   If ADDFLAG is off then
    ADDSUM equals MULTNUM2
   Else ADDSUM equals MULTNUM2 plus one
   Subtract 10 from ADDSUM
   If ADDSUM is negative then
    Begin
     Add 10 to ADDSUM
     Store ADDSUM in MULTNUM1
     Turn ADDFLAG off
    End of if ADDSUM is negative
   Else if ADDSUM is positive then
    Begin
     Store ADDSUM in MULTNUM1
     Turn ADDFLAG on
    End of if ADDSUM is positive
   Create a new block in MULTNUM1
   Set the pointer of the old block to point to the new block
  End of while MULTNUM2 is not at the end of the list
```

```
   If the carrry flag, ADDFLAG, is still on then
    Begin
     Store a one in MULTNUM1
     Create a new block in MULTNUM1
     Set the pointer of the old block to point to the new block
    End of if ADDFLAG is still on
   Store the a nil pointer in the last block of MULTNUM1
  End of the procedure ADD_NUMS
```

Procedure MODULUS will find the remainder of MODNUM1 divided by MODNUM2. The procedure will do long division to obtain the remainder. This is done by setting the two list to their tail pointer and then go backwards through the list until the end of MODNUM1 is found. Note that if the end of MODNUM2 is found before the end of MODNUM1 then the procedure will stop since MODNUM2 is already smaller then MODNUM1. Once the numbers are arranged correctly then MODNUM1 will be subtracted from MODNUM2 until the left part of MODNUM2 is smaller then MODNUM1. Next, the next digit to the right in MODNUM2 is brought down to form a new number to subtract MODNUM1 from. Once the right most digit in MODNUM2 is used and MODNUM2 is smaller then MODNUM1, the remainder is what is left in the list MODNUM2. The procedure uses doubly link lists to enable the procedure to go in either direction in the list, frontwards or backwards.

```
Begin
  Create new block in MODNUM2
  Set the head pointer to this block
  Call the procedure LIST_TO_DBL_LIST to make the singly link list
   given into a doubly link list
  Set MODNUM1 & MODNUM2 to the tail of the list
  While MODNUM1 is not at the beginning of the list do
   Begin
    If MODNUM2 is not at the beginning of the list do
     Back MODNUM1 & MODNUM2 back one block
    Else stop the procedure
   End of while not at the head of MODNUM1
  Advance MODNUM1 & MODNUM2 one block
  Set CURRENTMOD2 to the current position in MODNUM2
  Set the flag to end the subtraction routine, ENDFLAG, to off
  While ENDFLAG is off do
   Begin
    While the end of MODNUM2 is not reached and the flag to check
     if there is a -1 result produced, MODFLAG, is off do
     Begin
      If MODNUM1 is less then MODNUM2 then
       Subtract MODNUM1 from MODNUM2
      Else if MODNUM1 is larger then MODNUM2 then
       Begin
        If the next block in MODNUM1 is not the end of the list and
         not a zero then
         Begin
          Add 10 to MODNUM2
          Subtract one from the next block in MODNUM2
          Subtract MODNUM1 from MODNUM2
```

```
    End of if the next block in MODNUM1 is not the end of list
   Else if the next block in MODNUM2 is a zero then
    Begin
     Set a dummy pointer to the current block in MODNUM2
     Advance MODNUM2 t the next block
     While MODNUM2 is a zero do
      Begin
       Change the zero in MODNUM2 to a 9
       Advance MODNUM2 to the next block
      End of while MODNUM2 is a zero
     Subtract one from the current block in MODNUM2
     Reset MODNUM2 to the original position before this routine
    End of if the next block in MODNUM2 is a zero
   Else if the next block in MODNUM2 is the end of the list then
    Begin
     Set a dummy pointer to the current position in MODNUM2
     Subtract MODNUM1 from MODNUM2
     If the result is a -1 then
      Begin
       Set the next block in MODNUM2 to zero
       If the next block after the zero is not the nil
        pointer then set that block to be -1
       Else if the next block after the zero is the nil
        pointer then
        Begin
         Set that block to be -1
         Set a dummy pointer to that position in MODNUM2
         Create a new block in MODNUM2
         Store a -2 nil pointer in that block
         Set the back pointer of that block
         Set the front pointer of the previous block
       End of else if the next block after the zero is the nil
     End of if the next block in MODNUM2 is the end of the list
  End of if MODNUM is larger then MODNUM2
  Advance MODNUM1 and MODNUM2 to the next block
  Call procedure UPDATE to up date the end of the list
 End of while the end of MODNUM2 has not been reached
Set MODNUM1 & MODNUM2 to the start of the list
Set the flag to check for a carry in addition to off
While the end of MODNUM2 has not been reached do
 Begin
  If the addition carry flag, MODFLAG, is off then
   Add MODNUM1 & MODNUM2
  Else if there is a carry, MODFLAG is on, then
   Add MODNUM1, MODNUM2, and one
  Subtract 10 from the previous sum and store in MODFLAG
  If MODFLAG is positive then
   Begin
    Set MODFLAG to the on position
    Store the sum of MODNUM1 & MODNUM2 minus 10
  End of if MODFLAG is positive
  Else turn MODFLAG offf
  Advance MODNUM1 & MODNUM2 to the next block
 End of while the end of MODNUM2 has not been reached
```

```
    If the next block in MODNUM2 is a zero then
     Begin
      If MODFLAG is on then
       Add the current block in MODNUM1 & MODNUM2
      Else if MODFLAG is on then
       Add the current block in MODNUM1 & MODNUM2 plus one
      Turn MODFLAG off
     End of if the next block in MODNUM2 is a zero
    Set MODNUM1 to the nil pointer
    Call procedure UPDATE to update the list MODNUM2
    Set CURRENTMOD2 to the next block to the right in MODNUM2
    Set MODNUM1 to the start of the list
    Set MODNUM2 to CURRENTMOD2 in the list
    If the end of the list MODNUM2 has not been reached then
     Set the flag to stop the long division routine to off
    Else if the end of the list has been reached then
     Set the flag to stop the division routine to on
   End of while the flag to stop the division is off
  Set the lists MODNUM2 and RESULT to the start of the list
  Dispose of the list RESULT
  Create a new block in the list RESULT
  Set the head of the list to the new block
  While the end of the list MODNUM2 has not been reached do
   Begin
    Store the current block of MODNUM2 in RESULT
    Set a dummy pointer to the current block in RESULT
    Create a new block in RESULT
    Set the pointer of the old block to point to the new block
    Advance MODNUM2 to the next block in the list
   End of while the end of the list MODNUM2 has not been reached
  Store a nil pointer in the last block of RESULT
  Dispose of MODNUM2 list
End of the procedure MODULUS
```

Subprocedure UPDATE will delete any zeros that are at the end of the number being mod. This is done by going through the list until the end of the list is found and then go bachwards replacing zeros with -1. The procedure will not up date if the modulus is not at the end of the list and there was a -1 result produced by the division routine.

```
Begin
  If MODNUM1 is not at the end of the list and there was not a -1
   result in the divsion routine then
   Begin
    Reset MODNUM1 to the start of the list
    Go through the number being mod until the nil pointer is found
    While MODNUM2 is a zero do
     Begin
      Store a -1 for the zero in MODNUM2
      Back MODNUM2 to the next block in list
     End of while MODNUM2 is a zero do
    Reset MODNUM2 to the position in the list before updating
   End of if MODNUM1 was not at the end of the list
End of the procedure UPDATE
```

INTRODUCTION PROCEDURES

Procedure INTRO will first ask the user for their code number. If the code number is real then INTRO will write the greetings. If the code number is bogus then the program will terminate itself. After the greetings the user will be asked if they are ciphering or deciphering. According to the user response the ciphering or deciphering key will be used. Also a flag is set for the rest of the program to indicate waether to use a ciphering or deciphering routine.

```
Begin
  Ask user for code number
  Call procedure CHECKNUM to check the code number
  If the code number is bogus then end the program
  Else write the greetings and instructions
  Ask the user if they are ciphering of deciphering
  If they are ciphering then
   Set the variable CIPHORDECIPH to C
  Else set CIPHORDECIPH to D
  Call procedure SETCODES to read in the primes and coding keys
  Create new block in the modulus linked list, MODNUM1
  Set the head of the list to the new block
  Call prcedure LIST_TO_DBL_LIST to put the sum of the two primes
   in a doubly linked list
  Dispose of the first prime list
End of the procedure INTRO
```

Subprocedure CHECKNUM will use the inputted code number and check to make sure it is a real code number. If the inputted number is not in the list of codes then a negative answer will be returned to the INTRO procedure and INTRO will terminate the program.

```
Begin
  Reset the text file to be used
  Set the variable to contain the answer
  While the answer is not yes and the end of the text file is not
   found do
   Begin
    Read the current line of the text
    If the inputted code number and the code number just read
     are not equal to each other then
     Set the answer to no
   End of while the answer is not yes
  If the answer is not yet then set the answer to no
End of the procedure CHECKNUM
```

Procedure SETCODES will read through the file that contains the code numbers, primes, and keys until the inputted code number mathes with the code number in the text file. Then SETCODES will use that set of primes, P & Q, and either the ciphering or deciphering key for this run of the program. The two primes will be multiplied together to produce the modulus and the key will be used for the binary table.

```
Begin
  Reset the file that contains the code numbers, primes, and
   coding keys
  Read the first line in the text
  While the proper code number is not found and the end of the
   file is not found do
   Begin
    Read past the primes and coding keys to the next code number
    Read the next code number
   End of while the proper code number is not found
  If the code number was not found then end the program
  Set the flag to tell the end of the prime to off
  Create a new block in the list PRIME1
  Set the head pointer to this block
  Store a nil pointer of -1 in block
  Read the first digit in the first prime
  While the flag for the end of the prime is off do
   Begin
    Create a new block in PRIME1
    Store the first digit in the new block
    Set the new block to point to the old block
    Set the head of the list to the current block
    Read the next digit in the prime
    If the next digit is a blank then
     Turn the flag for the end of the prime on
   End of while the flag for the end of the prime is off
  Repeat the same process for the second prime and store it
   in PRIME2
  If the user is ciphering then
   Read the first key in the text
  Else advance the text file to the next line and
   Read the second key in the text
  Call procedure MULT_NUMBERS to multiply the two primes together
  Dispose of the link list PRIME2
End of the subprocedure SETCODES.
```

Writing Procedures

Procedure WRITE_CIPHER_TEXT will write the ciphered message to the text file NEWTEXT. This procedure will be called after each block has been coded.

```
Begin
  Set the list containing the coded block to start
  While the end of the coded block list is not found do
   Begin
    Write the current digit in the list to the file NEWTEXT
    If the end of the line in the text file is found then
     Reset the text file the next line in the file
   End of while the end of the coded block list is not found
  Write a end of number marker to the file
  Check for the end of the line agian
End of the procedure WRITE_CIPHER_TEXT.
```

Procedure WRITE_DECIPH_TEXT will write the decoded block to the printer. The decoded message will be changed from a number into two set of blocks and then changed to a character.

```
Begin
  Reset the list that contains the decoded block
  Separate number into two blocks
  Change into characters
  Write both characters to the printer
  If the end of the line has been found then
   Reset the printer to the next line
End of the procedure WRITE-DECIPH_TEXT
```

MAINLINE

The main line of the program wll first call INTRO to verify the code number and read, in the primes and the cipher and decipher keys. Then, the cipher or decipher number will be put into a binary number and all input and output files will be reset. Next, a block of data is read, coded and written. This is continued until the end of the file being read is found.

```
Begin
  Call procedure INTRO
  Call procedure BINARYCON
  Reset the files being used
  While the end of the file being read is not found do
   Begin
    Call reading procedures
    Proform Algorithm
    Call writing procedures
   End of while the end of the file being read is not found
```

APPENDIX 1

VARIABLE LISTING

PROGRAM CRYPTOSYS:

Labels: 666:The position in program for a goto statement. A goto statement is used if the user tries to input a bogus code number.

Constants:
- BINMAX:The maximum length for the Binarray Table.
- ENDOFLINE: This is the symbol for the end of a line marker.
- SIZE: This is one less then the number of characters read when ciphering.
- LINESIZE: The size of each sent to the text file or the printer.

Type:
- POINTER=PNTRECORD: Creates a pointer of type PNTRECORD.
- PNTRECORD: A record that contains an integer and a pointer.
- NUMBER: integer of PNTRECORD.
- NEXT: Pointer of PNTRECORD.
- DBLPOINTER=DBLRECORD: Creates a pointer of type DBLRECORD.
- DBLRECORD: A record that contains an integer and two pointers.
- NUMBERDBL: Integer of DBLRECORD.
- DBLNEXT: One pointer of DBLRECORD.
- DBLLAST: Second pointer of DBLRECORD.

Variables:
- CODES: File that contains the code numbers.
- CODENUMS: File that contains the code numbers, first prime, second prime, cipher number, and decipher number.
- OLDTEXT: File that will be used to cipher or decipher.
- NEWTEXT: File that will get the coded message.
- RESULT: Link list that contains code block for the algorithm.
- RESULTHEAD: Head of link list RESULT.
- BASE: Link list that contains original code block.
- BASEHEAD: Head of the link list Base.
- DUMRESULT: Dummy link list used to square RESULT.
- DUMRESULTHEAD: Head of link list DUMRESULT.
- DUMPOINTER: Dummy pointer.
- MODNUM1: Modulus link list.
- MODHEAD: Head of link list MODNUM1.
- MAINGEN: Generates binary number in reverse order.
- BINARRAY: Contains binary number.
- BINCOUNT: Contains the number of digits in binary number.
- POWER: Contains either the cipher or decipher number.
- CIPHORDCIPH: Flag used to see if user is ciphering or deciphering.
- MAINFLAG: Flag used to show end of file if found during READCIPHTEXT or READDECIPHTEXT.

LINEVAR: A variable used to keep the current position of the line in the text file or printer.

Procedure INTEGER TO LINK LIST
Variables:
Global:
RESULT: Cipher block number.
BASE: Original cipher block number.
RESULTHEAD: Head of result list.
BASEHEAD: Head of base list.
LISTNUM: The number read for ciphering.

Local:
PUTVAR1: Original integer being put in to list.
PUTVAR2: Size of original integer in power of 10's.
PUTGEN: A FOR statement variable.

Called By: READCIPHTEXT

Procedure CREATE DUMMY LIST
Variables:
Global:
DUMPOINTER: Dummy pointer.

Local:
CDLNUM1: Original link list.
CDLHAED1: Head of original link list.
CDLNUM2: Duplicate link list of original.
CDLHEAD2: Head of CDLNUM2 link list.

Called By: READDCIPHTEXT,MAINLINE

Procedure LIST TO DBL LIST
Variables:
Local:
DBLNUM1: Singly link list to be used.
DBLHEAD1: Head of DBLNUM1 list.
DBLNUM2: Doubly link list being created.
DBLHEAD2: Head of DBLNUM2 list.
DUMDBL: Dummy doubly link list pointer.

Called by: INTRO, MODULEST

Procedure READCIPHTEXT
Variable:
Global:
RESULT: Cipher block number.
BASE: Original base of cipher block.
RESULTHEAD: Head of RESULT list.
BASEHEAD: Head of BASE list.
MAINFLAG: A flag used to tell that the end of the file has been reached but it was found in the middle of

a line.
ENDOFLINE: End of line constant.

Local:
CHAR1: First character read.
CHAR2: Second character read.
REPROD: Product of ORD(CHAR1) times 100.
RESUM: Sum of first and second character digit value.
READFLAG: Flag that checks for end of file.

Called by: MAINLINE

Procedure READDECIPHTEXT
Variables:
Global:
RESULT: Cipher block number.
BASE: Original base to cipher block.
RESULTHEAD: Head of RESULT list.
BASEHEAD: Head of BASE list.
MAINFLAG: Flag used to tell that the end of the file has been found in the middle of a line.

Local:
REDNUM: ORD of REDCHAR-48.
REDCHAR: Character read.

Called by: MAINLINE
Calls: CREATE_DUMMY_LIST

Procedure BINARYCON
Variables:
Global:
BINARRAY= Array that holds the binary number.
BINCOUNT= The number of digits in BINARRAY.

Local:
BINVAR: The result of original number (1 or 0).
BINPOW: Original decimal number.

Called by: MAINLINE

Procedure MULT_NUMBERS
Variables:
Global:
DUMPOINTER: Dummy pointer.

Local:
NUMBER1: Number to be multiplied with NUMBER2.
NUMBER2: Number to be multiplied with NUMBER1.
NUMHEAD1: Head of NUMBER1 list.
NUMHEAD2: Head of NUMBER2 list.
DUMPNTMULT: Dummy pointer.
MULTNUM1: Result of each addition.

MULTHEAD1: Head of MULTNUM1 list.
MULTNUM2: Result of each multiplication to add to MULTNUM1.
MULTHEAD2: Head of MULTNUM2 list.
MCFLAG: Flag used to see if there is a carry on the last number in the list being multiplied.
MULTFLAG1: Holds the carry from the multiplication if any.
MULTFLAG2: Flag that used to see if zeros should be imbedded in MULTNUM2 list.
MULTVAR2: Represents the number of zeros needed for imbedding.
MULTGEN: Generates the number of zeros needed.
MULTPROD: Product of each multiplication between MULTNUM1 and MULTNUM2.
MULTMOD: Result of multiplication as a single digit.
MULTCOUNT: Is used to see if it is the first pass of procedure and to see if the end of the procedure has been reached.

Called By: MAINLINE
Calls: ADD_NUMS

Subprocedure ADD_NUMS
Variables:
Global:
MULTNUM1: Number to be added to MULTNUM2 and will hold the result.
MULTNUM2: Number to be added to MULTNUM1.
MULTHEAD1: Head of MULTNUM1 list.
MULTHEAD2: Head of MULTNUM2 list.
DUMPOINTER: Dummy pointer.

Local:
ADDSUM: The sum of each block of MULTNUM1 and MULTNUM2.
ADDFLAG: Flag for carry over of each addition if any.

Called By: MULT_NUMBERS

Procedure MODULEST
Variables:
Global:
RESULT: Cipher block number.
RESULTHEAD: Head of RESULT list.
MODNUM1: Modulus.
MODHEAD1: Head of MODNUM1 list.

Local:
MODNUM2: Doubly link list of RESULT list.
MODHEAD2: Head of MODNUM2 list.
DUMDBLPNT: Dummy doubley link list pointer.
MODFLAG: Flag to check if a -1 result occured or a carry in the addition routine occured.
CURRENT: Dummy pointer used to keep track of current list position.

Called By: MAINLINE
Calls: LIST_TO_DBL_LIST,UPDATE

Procedure INTRO
Variable:
Global:
MODNUM1: MODULEST.
MODHEAD1: Head of MODNUM1 list.
CIPHORDCIPH: Flag t see if the user is ciphering or deciphering.

Local:
CODENUMBER: Inputted code number.
ANSWER: Flag to see if code number inputted is real.
INTROCHAR: Inputted C or D for ciphering or deciphering.
PRIME1: First prime read from CODES.
PRIME2: Second prime read from CODES.
PRMHEAD1: Head of PRIME1 list.
PRMHEAD2: Head of PRIME2 list.

Called By: MAINLINE
Calls: CHECKNUM, SETCODES

Subprocedure CHECKNUM
Variables:
Global:
CODENUMBER: Inputted code number.
Local:
CHECKVAR: Code number read from CODENUMS.

Called By: INTRO

Subprocedure SETCODES
Variables:
Global:
PRIME1: First prime read from CODES.
PRMHEAD1: Head of PRIME1 list.
PRIME2: Second prime read from CODES.
PRMHEAD2: Head of PRIME2 list.
POWER: Power to raise cipher block.
DUMPOINTER: Dummy pointer.

Local:
SETVAR: Code number read from CODES.
SETCHAR: Digit of each prime read from CODES.

Called By: INTRO
Calls: MULT_NUMBERS

Procedure WRITE_CIPHER_TEXT
Variables:
Global:
RESULT: Coded block from ciphering.
RESULTHEAD: Head of RESULT list.

LINESIZE: A constant that tells the size of the line to be written.
LINEVAR: A variable that keeps the position of the being written.

Local:
WRITEVAR: Digit of coded block to write to file.

Called By: MAINLINE

Procedure WRITE_DECIPH_TEXT
Variables:
Global:
RESULT: Decoded block from deciphering.
RESULTHEAD: Head of RESULT list.
LINESIZE: A constant used to tell when the end of the line has been reached.
LINEVAR: A variable used to tell the position of the line being written.

Local:
WRITVAR: Holds first digit of deciphered message.
WRITVAR1: Holds second digit of deciphered message.
WRITCHAR: The character value of deciphered digits.

Called By: Mainline

MAINLINE

CALLS:
INTRO
BINARYCON
READCIPHTEXT
READDECIPHTEXT
CREATE_DUMMY_LIST
MULT_NUMBERS
MODULEST
WRITE_CIPHER_TEXT
WRITE_DECIPH_TEXT

APPENDIX 2

FILE STRUCTURE

The file that will contain the code numbers to the code files will look like this:

```
1430
4693
8167
 :
 :
```

Code file will contain a code number, two primes, and a cipher and decipher number. File will look like:

```
Code number: 1430
Prime 1:     937535832758
Prime 2:     387453852854
Ciph/Deciph: 158 1856
             4693
             2748778752385738
             9545235239535
             128 637
             8167
             34345539895
             85839829459
             485 9845
               :
               :
```

APPENDIX 3

ASCII CODE FOR THE CHARACTERS

A-B=65-97
a-z=97-122
 (*note that c=01, d=02,....z=24)
1-9=48-57