Cryptography

J.R. Hass

Introduction

Cryptography is the art of secret writing. If one must send confidential information to some destination, then most likely he does not want others to have access to it. This is where cryptography comes in. If we could somehow manipulate the information, so that it didn't matter if anyone saw the data besides those who were suppose to in the first place, then we wouldn't have to worry about the sensitive data being compromised.

Some key terms used in this presentation are:

Encryption: This is the act of hiding the data. There are many methods of manipulating data from one form to another so we will investigate a few.

Decryption: This is the act of changing the manipulated data back to its original form. The whole idea of hiding the information is so that eventually, someone will be able to see it.

Key: This is the piece of information that only those who are suppose to view the sensitive data have. It allows them to decrypt the encrypted data.

```
> restart: with(StringTools): with(numtheory): with(combinat):
Warning, the assigned name Group now has a global binding

Warning, the protected name order has been redefined and unprotected

Warning, the protected name Chi has been redefined and unprotected

>
```

Encryption

Caesar

Caesar encryption is a relatively simple one. It involves shifting letters of the alphabet. For example, if we moved every letter of the alphabet 1 to the right we would get a mapping like:

A -> B
B -> C
Z -> A

Etc.

Using frequency analysis, this encryption becomes quite trivial to break. This should be intuitive with a little thought. One could easily calculate the frequencies of all the letter of a large piece of English text. They could then look at the cipher text and do the same thing. If they noticed there were many 'T's for example, they could then guess that E was mapped to T. We will look at a modification to this cipher next to make it a bit more complex.

In my implementation, I shift according the ASCII table.

```
> Caesar := proc(plainText::string, offset::integer)

# An example of Caesar encryption. This method just shifts
# the data according to the ASCII chart by an offset given
# as a parameter.
#
# Params: plainText - The plain text to encrypt.
#         offset    - The number to shift the text.
#
# Returns: A list of the encrypted text and the offset to use for
#          decryption.

local cipher, i :

cipher := "" :

  for i in plainText do
    #
    # Since I use the mod operator, the offset can be anything!
    #
    cipher := cat(cipher, Char((Ord(i)+offset) mod 128)) :
  end do :

[cipher, offset]:

end proc:
> Caesar("Hello World, I am finally done with Finals!", 10);
   ["Rovvy*ay|vn6*S*kw*psxkvv_*nyxo*_s~r*Psxkv}+", 10]
> Caesar("We can even make the offset negative :-)", -50);
   ["%3n1/<n3D3<n;/93nB63n=44A3Bn<35/B7D3n\b{w", -50]
>
```

Vigenere

Vigenere is a slightly more complicated version of Caesar. The idea behind this encryption method is that you shift multiple letter of the plain text. This is done using a key that is provided by the user. Using this multiple shift idea, we can prevent people who try to use letter frequencies (explained above) to crack our code.

```
> Vigenere := proc(plaintext::string, codeword::string)

  # This is a program to encrypt a message using
  # vigenere encryption. Notice the input restrictions
  #
  # Params: plaintext - The text to encrypt. Notice that for this
  #                     I do not use the ASCII table for shifting,
  #                     just the alphabet. Input must be capitol
  #                     letters and no numbers or spaces.
  #
  #
  #         codeword  - The string to shift for each letter, must also
be
  #                     all capitols..
  #
  # Returns: A list containing the encrypted text and the key word for
  #          decryption.

  local pLen, cLen, n, cipher, i, shift:

  pLen := length(plaintext): cLen := length(codeword):

  # Check if the length of the plain text is a multiple of the
  # length of the code word, if its not adjust n by the difference.

  if pLen mod cLen <> 0 then
    n := pLen + (cLen - (pLen mod cLen)):
  else n := pLen:
  end if:

  # Shift each letter of the plain text with a corresponding letter
  # of the code word.

  cipher := "":
  for i from 1 to n do
    cipher := cat(cipher, Char((Ord(plaintext[(i-1) mod pLen+1]) +
                               Ord(codeword[(i-1) mod cLen+1])) mod 26
+ 65)):
  end do:

  [cipher, codeword]:

  end proc:
> Vigenere("APICTUREISWORTHATHOUSANDWORDS", "BREAKME");
   ["BGMCDGVFZWWYDXIRXHYGWBEHWYDHTRTIMFY", "BREAKME"]
> Vigenere("ITISFINALLYTIMEFORSUMMERTOBEGIN", "MATH");
      ["UTBZRIGHXLRAUMXMARLBYMXYFOULSIGP", "MATH"]
> Vigenere("EVERYONEATUCSDSHOULDTAKEACOURSEONMAPLE", "CARBON");
["GVVSMBPERUIPUDJICHNDKBYRCCFVFFGOENOCNEVWSE", "CARBON"]
>
```

Rectangular Transposition

Rectangular transposition uses a whole different idea of encoding. The idea here is to first

write your plain text in rectangular form, literally. For example consider the plaintext:

H I T H E R E
H O W A R E U

Now, we have 7 columns of text. If we permuted the numbers 1-7 we could then pick out random columns to write our message with. Say we picked the permutation of:

1, 3, 5, 7, 2, 4, 6 … Then our message would be

HHTWEREUIOHARE.

```
> RectTrans := proc(plainText::string, n::integer)

  # This method encrypts the plain text according to the rectangular
  # transposition method.
  #
  # Params: plainText - The text to be encrypted. Plain text is expeted
  #                     to be all capitols.
  #         n         - The number to permute to generate the
  #                     encrypted text.
  # Returns: A list of the encrypted text and the permuted numbers.

  local tempSet, pList, temp, randGen, i, numRows, strLen,
        j, cipher:

  randGen := rand(n):
  tempSet := {}: pList := []:

  # Generate a random permutation of numbers from 1-n. We use the fact
  # that sets in maple contain unique elements to know when to stop
adding
  # random numbers.

  for i from 1 to n do
    while(nops(tempSet) <> i) do
      temp := randGen() + 1:
      tempSet := {op(tempSet), temp}:
    end do:

    pList := [op(pList), temp]:
  end do:

  # Check to make sure the plaintext can be written as a rectangle.

  strLen := length(plainText):
  if(strLen mod n = 0) then numRows := strLen / n:
  else numRows := floor(strLen / n + 1):
  end if:

  # After we write the plaintext as a rectangle, we go through the
  # columns according to our random permutation and record the letters
  # in that column. If we encounter the end of the plaintext, we just
  # use the random letter Q, this is obviously arbitrary.
```

```
  cipher := "":
  for i from 1 to n do
    temp := pList[i]:
    for j from 1 to numRows do
      if(temp > strLen) then
        cipher := cat(cipher, "Q"):
      else
        cipher := cat(cipher, plainText[temp]):
      end if:
      temp := temp + n:
    end do:
  end do:

  [cipher, pList]:
  end proc:
> RectTrans("INCITISEASYTOSHOOTYOURSELFINTHEFOOT", 5);
["NSTORIFIIYOUFECEOTSNOIASYETOTSHOLHT", [2, 1, 3, 4, 5]]
>
RectTrans("CPPMAKESITHARDERTOSHOOTYOURSELFBUTWHENYOUDOYOUBLOWOFFYOURLEG
", 8);
 ["SRYBOLUQMAHSHYFGPTOUTDWLCITOUUORPHSRWOOEKDOLNUYQAROEEOFQEETFYBOQ",
[8, 4, 2, 1, 3, 6, 5, 7]]
>
```

ADFGVX

The ADFGVX is quite a complicated system. It begins by having a keyword and an even number that will be permuted. You start by generating a matrix which looks like as follows:

	A	D	F	G	V	X
A	k	e	y	w	o	r
D	d	a	b	c	f	g
F	h	i	j	l	m	n
G	p	q	s	t	u	v
V	x	z	0	1	2	3
X	4	5	6	7	8	9

So the keyword begins the matrix and then you fill it up with the rest of the alphabet and zero through nine. Now each letter has a coordinate system of the ADFGVX matrix. For example, the letter 'r' would be AX.

Now we can write out our text in columns of N/2, N being the even number that you passed to this method. For each letter now, we can get a set of coordinates. We now have N columns of letters after getting two coordinates for each plain text letter.

Using rectangular transposition, we can encode the resulting text of A's D's F's G's V's and X's. This will then only give us cipher text containing the letters of ADFGVX.

```
> Index := proc(x, l::list)
  # Helper method that returns the index of the element x in the
  # passed list.

  local i, n;
  n := nops(l);
  for i from 1 to n do if l[i] = x then return i; end if; end do;
end proc:
> LookUp := proc(x, matrix::list)
  # Helper method to return us the coordinates of the passed element
  # in the ADFGVX matrix.

  local i,j;
  for i from 1 to 6 do
    if member(x, matrix[i], 'j') then return [i,j]; end if;
  end do;

  error("Plain text must be a capitol letter or 0-9");
end proc:
> MakeMatrix := proc(codeword::string)
# Makes the ADFGVX matrix with the codeword passed to it
# as explained above.

  local i,j, templist, matrix, len, n, l;

  templist := [];
  for i from "A" to "Z" do templist := [op(templist), i]; end do;
  for i from 1 to 10 do templist := [op(templist), convert(i-1,
string)];
  end do;

  matrix := []; len := length(codeword); n := 1;
  for i from 1 to 6 do
    l := [];
    for j from 1 to 6 do
      if n <= len then
        if member(codeword[n], templist) then
          l := [op(l), codeword[n]];
          templist := subsop(Index(codeword[n], templist)=NULL,
                      templist);
          n := n+1;
        else
          l := [op(l), templist[1]];
          templist := subsop(1=NULL, templist);
          n := n+1;
        end if;
      else
        l := [op(l), templist[1]];
        templist := subsop(1=NULL, templist);
```

```
      end if;
    end do;
    matrix := [op(matrix), l];
  end do;
  matrix;
end proc:

> ADFGVX := proc(plainText::string, N::integer, codeword::string)

  # This progrm encrypts the text with the ADFGVX encryption.
  #
  # Param: plainText - The text to encode.
  #        N         - The integer to permute with the ADFGVX encryption
  #        codeword  - The word to use to help generate the ADFGVX matrix.
  #
  # Note: This method expexts all text to be capitol letters and numbers.
  #
  # Returns: A list of the encrypted text, the codeword, and the permuted set
  #          of numbers.

  local letter, cipher, matrix, len, n, i, l;

  if N mod 2 <> 0 then error("N must be an even number!");  end if;

  matrix := MakeMatrix(codeword);
  n:= length(plainText);

  if n mod N/2 <> 0 then
    n := n + (N/2- (n mod N/2));
  end if;

  len := length(plainText);
  letter := ["A", "D", "F", "G", "V", "X"];
  cipher := "";
  for i from 0 to n-1 do
    l := LookUp(plainText[(i mod len)+1], matrix);
    cipher := cat(cipher, letter[l[1]]);
    cipher := cat(cipher, letter[l[2]]);
  end do;

  l := RectTrans(cipher, N);
  [l[1], l[2], codeword];
end proc:
>
ADFGVX("FEWARETHOSEWHOSEEWITHTHEIROWNEYSANDFEELWITHTHEIROWNHEARTSQEINST
EIN", 8, "SCOOBYDOO");

["ADGDGDGADGGGDGDDDDDAAGGGDFDGDGAGAFXFDAXVFXAFVXVDDFFFDFVDVXVGDXVFDAVXG
VXDFDXAFXFDVFXDDGGDA\

        DDAADFDDFGDGDDDAFXFDDVDFDXGAAVDGADDDDFADDDADAFD", [7, 3, 6, 2,
8, 5, 4, 1], "SCOOBYDOO"]
> ADFGVX("IDONTKNOWHOWANYONEFIGUREDOUTHOWTOCRACKTHIS", 12, "JRHASS");

["FAFGGDAFGVDGFGDVGDFDXXGGGDVAAVFFVGVVVAVFVDDFADDGDGFVFDAFDVGXVDADFFDFG
FFGDDFAAGFFAAAD",
```

```
        [7, 5, 10, 2, 12, 6, 1, 8, 4, 3, 11, 9], "JRHASS"]
>
```

RSA

The RSA code is still widely used today. It is also called public key cryptography. This code uses the fact that it is very difficult to factor large numbers as its strength in being unbreakable. Surprisingly, relative to some of the earlier codes, it is quiet easy to explain how this one works.

Begin by picking two primes p1 and p2. These numbers are generally very large. Multiply them together to get an m, so:

m = p1 * p2

Now, we know that since m is the product of two primes, then phi(m) is going to equal:

phi(m) = (p1-1) * (p2-1)

Now we pick any e and d such that

e*d = 1 mod phi(m) Note that it is integral that the e choosen is relatively prime to phi(m). The

reasoning behind this will be explained later.

e is going to be our encrypting key, d is going to be our decrypting key.

d is what you want to keep private from the public.

To generate the code, we need to first take our plain text and turn it into a representation of numbers. This can be done in any arbitrary way.

To encode, we simply raise the numbers of our plain text to the power of e and take the modulus of m.

So E1 = P1 ^ e mod m

This gives us our encrypted text. To decode we simply raise an encrypted number to the

power of d and take the modulus of m

P1 = E1 ^ d mod m

```
> RSA := proc(plainText::string, m_greaterThan::integer)

  # This procedure takes a string an encodes it using the RSA
  # encryption method.
  #
  # Params: plainTest     - The string to encode
  #         m_greaterThan - This will ensure that the number we choose
  #                         to take phi of will be the product of the
  #                         next two primes after this number.
  #
  # Returns: This method returns a list of the encrypted text
  #          and the 'e', 'd' and 'm' values used to decrypt the text

  local phiM, p1, p2, m, e, d, dummy, randGen, cipher, i,
        retVal;

  p1 := nextprime(m_greaterThan);
  p2 := nextprime(p1);
  phiM := (p1-1)*(p2-1);
  m := p1*p2;

  randGen := rand(phiM);
  e := randGen();

  # igcdex conveiently figurs out what 'd' is for us
  while igcdex(e, phiM, 'd', 'dummy') <> 1 or d < 0 do e := randGen();
end do;

  cipher := [];

  # Now we just raise the plaintext to the power of e and mod it m.
  # Notice the & operator used for large numbers.

  for i in plainText do
   cipher := [op(cipher), Ord(i) &^e mod m];
  end do;

  [cipher, e, d, m];

  end proc:
>
RSA("ICANTBELIEVEITTAKESSOLITTLECODETOWRITETHISPOWERFULENCRYPTIONMETHOD
", 1000);
  [[974625, 780057, 352236, 809104, 728719, 781436, 107027, 304770,
974625, 107027, 414359,

        107027, 974625, 728719, 728719, 352236, 122291, 107027, 501051,
501051, 222175, 304770,

        974625, 728719, 728719, 304770, 107027, 780057, 222175, 868136,
```

```
107027, 728719, 222175,

        34978, 86692, 974625, 728719, 107027, 728719, 341979, 974625,
501051, 948810, 222175,

        34978, 107027, 86692, 315801, 718934, 304770, 107027, 809104,
780057, 86692, 690414,

        948810, 728719, 974625, 222175, 809104, 321073, 107027, 728719,
341979, 222175, 868136],

        805813, 507613, 1022117]
>
RSA("LETSUPTHESIZEOFMALITTLEBITANDSEEHOWLONGITTAKESMAPLETODOTHECOMPUTAT
IONS", 2^20);
  [[143006057764, 815676470467, 127160894710, 888760637646,
1054921107533, 671668224542,

        127160894710, 1049933171982, 815676470467, 888760637646,
252520175739, 185949705342,

        815676470467, 507670240547, 467159188463, 103450703643,
57371389111, 143006057764,

        252520175739, 127160894710, 127160894710, 143006057764,
815676470467, 560846061432,

        252520175739, 127160894710, 57371389111, 115667106036,
704578425924, 888760637646,

        815676470467, 815676470467, 1049933171982, 507670240547,
1072224366698, 143006057764,

        507670240547, 115667106036, 987773105173, 252520175739,
127160894710, 127160894710,

        57371389111, 311404227395, 815676470467, 888760637646,
103450703643, 57371389111,

        671668224542, 143006057764, 815676470467, 127160894710,
507670240547, 704578425924,

        507670240547, 127160894710, 1049933171982, 815676470467,
55025816164, 507670240547,

        103450703643, 671668224542, 1054921107533, 127160894710,
57371389111, 127160894710,

        252520175739, 507670240547, 115667106036, 888760637646],
759060027323, 309386620403,

        1099532599387]
>
```

Decryption

Caesar

To decrypt the Caesar code, it is actually quite trivial. All we have to do is to shift back the text the other way! Example:

```
> CaesarDecrypt := proc(eText::string, offset::integer)

  # This method decrypts a Caesar encrypted message
  #
  # Params: eText  - The encrypted text
  #         offset - The offset it was encrypted with
  #
  # Returns: The original plain text

  local n;

  # Just shift backwards!

  n := Caesar(eText, -offset);
  n[1];
end proc:
> e := Caesar("Gravity cannot be held responsible for people falling in
love - Einstein", 100);
     e :=
["+VEZMX]_GERRSX_FI_LIPH_VIWTSRWMFPI_JSV_TISTPI_JEPPMRK_MR_PSZI___)MRWX
IMR", 100]
> CaesarDecrypt(e[1], e[2]);
 "Gravity cannot be held responsible for people falling in love -
Einstein"
>
```

Vigenere

The same idea is used for Vigenere, except now we have to shift a different amount for each letter that is in the key (the codeword that was passed to the original function).

```
> VigenereDecrypt := proc(eText::string, key::string)

  # This method decrypts a message that was encrypted with the
  # Vigenere method of encryption
  #
  # Params: eText - The encrypted text
  #         key   - The key to decrypt with
  #
  # Returns: The plain text of the message

  local n, i, pText, cl, kLen;

  pText := "";
  n := length(eText);
  kLen := length(key);

  # Now shift according to the letter the plaintext was mapped to
  # in the encrypting codword.
```

```
  for i from 1 to n do
    cl := Ord(eText[i]) - 65;
    cl := cl - Ord(key[(i-1) mod kLen +1]);

    while cl < 65 do cl := cl + 26; end do;
    pText := cat(pText, Char(cl));
  end do;
  pText;
end proc:
> e := Vigenere("ICANNOTBELIEVEIAMASENIORINCOLLEGEWHEREDIDTHETIMEGO",
"BUGS");
    e := ["JWGFOIZTFFOWWYOSNUYWOCUJJHIGMFKYFQNWSYJAENNWUCSWHIOU",
"BUGS"]
> VigenereDecrypt(e[1], e[2]);
 "ICANNOTBELIEVEIAMASENIORINCOLLEGEWHEREDIDTHETIMEGOIC"
>
```

Rectangular Transposition

Rectangular transposition is a bit tricky to decrypt. Visually, one would want to make actual columns of text with the encrypted text. Then if they were given the permutation of the number that it was encrypted with, they can just put the columns in the correct order.

Our method is going to find the position of the column we are looking for in the permutation list. We are then going to search the string of text and find which letter in a given column we need to append to the plain text string.

```
> RectTransDecrypt := proc(eText::string, key::list)

  # This method decrypts a string of text encoded with
  # rectangular transposition.
  #
  # Params: eText - The encrypted text
  #         key   - The permutation used to encrypt the text
  #
  # Returns: The plain text message

  local rows, cols, pText, i, j, p;

  pText := "";
  cols := nops(key);
  rows := length(eText)/cols;

  for i from 1 to rows do
    for j from 1 to cols do
      member(j, key, 'p');
      pText := cat(pText, eText[rows*(p-1)+i]);
    end do;
  end do;
  pText;
end proc:
> e :=
```

```
RectTrans("WHENYOUSITWITHANICEGIRLFORTWOHOURSITSEEMSLIKETWOMINUTEWHENYO
USITONAHOTSTOVEFORTWOMINUTESITSEEMSLIKETWOHOURSTHATSRELATIVITY", 16);
  e :=
["TRLNVTRYTOEUOEHQEEINAIELAOWITSTQSFMHTSOIIOSEOIUTIWKOFETQWIRMOOIRNUOTW
LSQHHTSRMAQWTIY\

        ESSQNGTUHNTAYISTOUWTULEWSEHVOREETTOIHCSINMKE",

        [10, 13, 3, 15, 8, 9, 12, 1, 16, 14, 11, 4, 5, 7, 6, 2]]
> RectTransDecrypt(e[1], e[2]);

"WHENYOUSITWITHANICEGIRLFORTWOHOURSITSEEMSLIKETWOMINUTEWHENYOUSITONAHOT
STOVEFORTWOMINUTESIT\
        SEEMSLIKETWOHOURSTHATSRELATIVITYQQQQQQ"
>
```

ADFGVX

As with the encryption scheme, the decryption scheme of ADFGVX is also a bit tricky. ADFGVX uses rectangular transposition inside of it. So we must also use the method to decrypt a rectangular transposition string first, and then continue from there.

When we encrypted, we use an alphabet/number matrix to find 'coordinates' for each letter of the plain text. Now we will have to go back words, and for each pair of coordinates, find the corresponding letter to go with it.

```
> ADecrypt := proc(eText::string, pList::list, key::string)

  # This method decrypts a string that was encrypted with the
  # ADFGVX method.
  #
  # Params: eText - The encrypted text
  #         pList - The permutation of an even number used for
encryption
  #         key   - The key that was used to make the matrix.
  #
  # Returns: The original plain text

  local matrix, n, letter, x, y, i, pText, EText;

  EText := RectTransDecrypt(eText, pList);

  pText := "";
  matrix := MakeMatrix(key);
  n := length(eText);
  letter := ["A", "D", "F", "G", "V", "X"];

  # Now just look up the plaintextin the matrix.

  for i from 1 to n by 2 do
    member(EText[i], letter, 'x');
    member(EText[i+1], letter, 'y');
```

```
    pText := cat(pText, matrix[x][y]);
  end do;

  pText;
end proc:
> e :=
ADFGVX("MYHATGOESOFFTOTHEPERSONTHATWASABLETOFIGUREOUTHOWTOBREAKTHIS",
10, "PROGRAM");
  e :=
["VAFGDVGGFDVADADDGAAAADAGGAGADGGGGAFDDDGAFGFFAGDFAGVXAVVFDXXDXGDAVVGAF
VGADFDDXADFDFFA\

        AFVGFDXVGFDXAGADGADAGGAGVADDAGADDAAD", [8, 1, 9, 5, 2, 6, 10,
4, 7, 3], "PROGRAM"]
> ADecrypt(e[1], e[2], e[3]);
       "MYHATGOESOFFTOTHEPERSONTHATWASABLETOFIGUREOUTHOWTOBREAKTHISM"
>
```

RSA

RSA makes it very easy to decrypt with. As we noted earlier, we will need to raise the encrypted text to the inverse of e to get
the plain text again...

Why does this work though? As promised above, a little more in depth explanation of why this encrypytion works.

We know that e and d are inverses of the phi(m). We also know that for some number 'a' (encrypted text say)
that if gcd(a, m) = 1, then

Euler-Fermat:
a ^ phi(m) = 1 mod m

Going through the encryption and decryption process looks like:
(a^e)^d = a^(ed) = a^phi(m) * a = 1 * a = a

When we encrypt our text, we use a relatively prime number to raise the plaintext to. However, this does not guarantee that the output (a^e) will also be realtively prime to m. What happens then?

We also have another theorem which states:

For any primes p and q
a ^ ((p-1) * (q-1)) = 1 mod pq

How conveient! We know that m = pq and phi(m) is just (p-1) * (q-1). So no matter what number we use, raising it to the e and
then to the d to decrypt is like

a ^phi(m) * a^1 = 1 * a mod m ....and hence we have the RSA encryption method!

```
> RSADecrypt := proc(eText::list, d::integer, m::integer)

  # Decrypts text that was encoded with RSA.
  #
  # Parmas: eText - The encoded text
  #         d     - The power to raise the the text to.
  #         m     - The modulus of our original encryption
  #
  # Returns: The plain text

  local pText, n, i;

  pText := "";
  n := nops(eText);

  for i from 1 to n do
    pText := cat(pText, Char(eText[i] &^d mod m));
  end do;
  pText;
end proc:
>
> e := RSA("THEWHOLEISMORETHANTHESUMOFITSPARTS", 100);
  e := [[4528, 4644, 537, 3925, 4644, 420, 2950, 537, 6476, 7617, 9941,
420, 3599, 537, 4528,

        4644, 9862, 10321, 4528, 4644, 537, 7617, 3464, 9941, 420,
2217, 6476, 4528, 7617, 6417,

        9862, 3599, 4528, 7617], 3311, 191, 10403]
> RSADecrypt(e[1], e[3], e[4]);
"THEWHOLEISMORETHANTHESUMOFITSPARTS"
>
```

Breaking RSA

Since RSA is quite a popular method of encryption, I wanted to run a test on it. I thought it would be interesting to see what values of 'm' would really make Maple stretch its limits. I wanted to write a program that 'breaks' RSA. In other words given the 'e' and the 'm' used to encrypt some text, it would figure out phi of m and compute the 'd' value to decrypt the text. The user can use this program to test how long certain values of 'm' take to break.

```
> RSABreak := proc(eText::list, m::integer, e::integer)

 # Method to break RSA encryption.
 #
 # Params - eText: The encrypted text
 #        - m:     The 'm' value used to encrypt the text
 #          d:     The 'e' value used to encrypt.
```

```
 #
 # Returns: A list of the amount of time it took in seconds to break
this
 #          code and the plaintext.
 #

 local i, d, phiM, pText, dummy;

 i := time();

 # Get phi of m
 phiM := phi(m);

 # Calculate 'd'
 igcdex(e, phiM, 'd', 'dummy');

 # Just call the decryption method

 [time()-i, RSADecrypt(eText, d, m)];

end proc:
> e := RSA("LETSSEEJUSTHOWGOODMYRSABREAKPROGRAMWORKS", 10^14);
  e := [[372488860441345871142676746l, 673732881428058653498936476 2,

        430034188353227169709149566 3, 503465488753741755273386363 9,
503465488753741755273386363 9,

        673732881428058653498936476 2, 673732881428058653498936476 2,
165099690214880486477850898 9,

        800629579891420779855245389 6, 503465488753741755273386363 9,
430034188353227169709149566 3,

        583295504017544178130466088 5, 388713054348676276717714757 5,
549688911149229023771818395 3,

        771004374050237431074427464 1, 388713054348676276717714757 5,
388713054348676276717714757 5,

        248741772204022611162273460 5, 652020187786917214479834995 9,
581589077151895070481255210 0,

        816317359618656167731005110 0, 503465488753741755273386363 9,
751564111938128073468339949 5,

        141961333667618854035454332 1, 816317359618656167731005110 0,
673732881428058653498936476 2,

        751564111938128073468339949 5, 904681941768916079782020995 5,
993160785733825390842829287 9,

        816317359618656167731005110 0, 388713054348676276717714757 5,
771004374050237431074427464 1,

        816317359618656167731005110 0, 751564111938128073468339949 5,
652020187786917214479834995 9,
```

```
        549688911149229023771818395 3, 388713054348676276717714757 5,
816317359618656167731005110 0,

        904681941768916079782020995 5, 503465488753741755273386363 9],

        114556335081595291058049889 7, 176110226537131201292117253 3,

        100000000000098000000000207 7]
> d := RSABreak(e[1], e[4], e[2]);
d := [1.610, "LETSSEEJUSTHOWGOODMYRSABREAKPROGRAMWORKS"]
```