USING THE TI–92 PLUS TO INTRODUCE THE RSA CRYPTOSYSTEM

Robert T. Harger and Neil P. Sigmon

1. Introduction. With society becoming more and more reliant on digital and computing technology, the ability to transfer information in a secure and confidential fashion using cryptography, the art of secret message writing, has increased dramatically in importance. As this reliance increases in the future, people will benefit in having at least some basic knowledge of this important topic. However, most students, especially those majoring in liberal arts and humanity curriculums, have no formal training in any of the mathematics techniques that are used in cryptography. Much of this problem stems from the fact that many cryptographic techniques are too tedious to perform by hand and rely on computing technology not readily available to all students for performing realistic examples.

To alleviate this problem, the topic of cryptography has been integrated into our finite mathematics course. This course, which exposes students to topics such as linear equation applications, matrices, and mathematics of finance, is designed to introduce some of the many "real–life" applications of mathematics. Preliminary evaluations from students have shown a favorable response for integrating cryptography into this course. As a part of this topic, the RSA Cryptosystem, currently one of the most widely used cryptosystems, is introduced. This cryptosystem is simplistic in its application in that one has to only understand the concepts of exponentiation and modular arithmetic to implement it. However, realistic applications of this system require large integers, which makes it impractical to compute by hand. Fortunately, the algorithms necessary to perform the computations required for the RSA Cryptosystem can easily be programmed into a TI–92 Plus graphics calculator.

The purpose of this article is to demonstrate how the RSA Cryptosystem is integrated into our finite mathematics course. In particular, we discuss how the TI–92 Plus plays an integral part in performing the needed computations. As a point of information, every student is required to have a TI–92 Plus (or TI–89) for our course. We first give a discussion of some basic background mathematics.

2. Preliminary Concepts. The RSA Cryptosystem relies on concepts from number theory that are familiar to most students; exponentiation, prime numbers, greatest common divisors, and modular arithmetic. All the numbers we work with in this paper are nonnegative integers. When denoting exponentiation we will use the following common notation; if a is any real number and n is any natural number, then

$$a^n = \underbrace{a \cdot a \cdots a}_{\text{multiplied } n \text{ times}} .$$

A natural number, p, is said to be prime if its only divisors are 1 and p. Any number that is not prime is called a composite number. The greatest common divisor of two natural numbers a and b, written as $\gcd(a, b)$, is the largest natural number which divides both a and b with no remainder. One important fact is the greatest common divisor of any two prime numbers is always 1. This result can also be true for composite numbers. More generally, two natural numbers a and b are said to be relatively prime if $\gcd(a, b) = 1$. Modular arithmetic arises from the elementary technique of long division. When performing the long division $a \div b$, we continue the process until we obtain a quotient, q, and a remainder, r, where $0 \leq r < b$. The result of any long division, $a \div b$, can be expressed mathematically by the equation

$$\frac{a}{b} = q + \frac{r}{b}.$$

Multiplying both sides of this equation by b yields $a = qb + r$. The latter equality demonstrates a more general theorem known as the division algorithm. The division algorithm states that if a and b are natural numbers with $b \neq 0$, then there exists unique numbers q and r such that $a = qb+r$. This remainder provides the basis for modular arithmetic. We say r is congruent to $a(\text{mod } b)$, written as $r \equiv a(\text{mod } b)$, if r is the remainder of the division $a \div b$. In the context of modular arithmetic, the number b is defined as the modulus and the concept of "congruence" can be thought of as a generalization of the equality relation.

There are many properties involving modular addition and multiplication, which are unnecessary for the scope of this paper. However, it is essential that we discuss the concept of multiplicative inverses. Given two natural numbers a and b, with $a < b$, we say the natural number t is the multiplicative inverse of a with respect to the modulus b if $t < b$ and $(a \cdot t)(\text{mod } b) \equiv 1$. Multiplicative inverses only exist when the numbers a and b are relatively prime. In this case, the multiplicative inverse t of a is unique with respect to the modulus b.

3. Cryptosystem Description. The RSA Cryptosystem is named after its developers, Rivest, Shamir, and Adelman, who first published the system in 1978. The setup of this cryptosystem is described by the following steps:

1. Choose two prime numbers p and q and form the products $n = pq$ and $f = (p-1)(q-1)$.
2. Two relatively prime natural numbers e and d are determined where $(e \cdot d)(\bmod f) \equiv 1$. Note in this choice that $\gcd(e, f) = 1$.
3. Given a message M we want to send, we encipher the message by computing

$$M^e (\bmod n) \equiv C.$$

   Note that in order to recover the message properly, $M < n$.
4. To recover the message, we compute

$$C^d (\bmod n) \equiv M.$$

The RSA Cryptosystem is an example of a public key cryptosystem. The intended recipient of a message will initiate steps 1 and 2 of the process in that he or she will determine two prime numbers and compute the integers n, f, e, and d. The integers n and e are made public knowledge. Any person who wants to send a message (step 3) to the recipient will use n and e to encipher the message. Only the recipient can recover the message (step 4) since only he or she knows the deciphering exponent d. The security of the method is based on the inability of any intruder to factor the integer into a product of its prime factors p and q, hence preventing the formation of the quantity f and the recovery of d that the recipient forms in step 2.

4. TI–92 Plus Implementation. In the finite mathematics class, students are taught elementary techniques using small integer values for performing primality testing, integer factorization, greatest common divisor calculations, and modular arithmetic. In addition, students implement the RSA computations for small integer values to get a feel of how the cryptosystem works. However, these calculations quickly become impractical for larger values. The TI–92 Plus provides a valuable tool for overcoming this problem which we now demonstrate.

To perform primality testing, the built-in **isPrime** command is provided to determine if a given number is prime. As demonstrated in Figure 1, we see that 1709 and 12373 are primes while 1560001 is not prime.

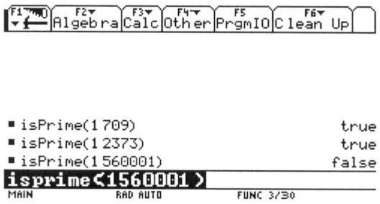

Figure 1

To generate prime numbers, we use the user-defined function **nextprim**. The code for this routine is given in the appendix. This function generates the smallest prime larger than the given input. As illustrated in Figure 2, we see that the smallest prime larger than 235441 is 235447. This can be verified by using the **isPrime** command. Next, we generate the primes 93652157, 780702347, 1093457277444666623, and 1234567890123456789012345678901234568123.

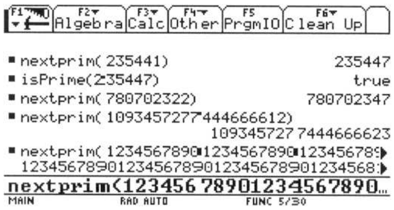

Figure 2

Three of the built-in features of the TI–92 Plus are **factor**, **gcd**, and **mod** commands. For example if we enter **factor**(7010254402661) the calculator will return $1234439 \cdot 5678899$. Entering **gcd**(5048,3271244) the calculator returns 4588. If we want 500234(mod 10301) we enter **mod**(500234,10301) and the TI–92 Plus returns 5786.

To demonstrate how the RSA Cryptosystem process works, suppose we use **nextprim** to generate a pair of primes. If we type **nextprim**(12340) and **nextprim**(35350) the calculator returns 12343 and 35353, respectively. We then store each of these (using the **STO** key) as p and q. We can now calculate n and f by entering $p \cdot q \to n$ and $(p-1) \cdot (q-1) \to f$.

To assist in completing step 2 of the RSA process, the user–written function **inverse** (the code for this routine is given in the appendix) is provided. Suppose we use **nextprim** to choose $e = 92347$. Figure 3 demonstrates how the deciphering exponent $d = 5094211$ is found with the inverse routine by using $e = 92347$ and the modulus $f = 436314384$ as parameters.

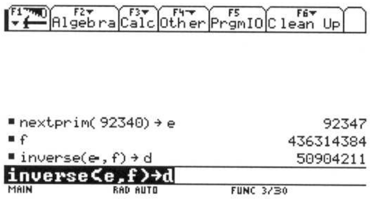

Figure 3

We should note that e and d are multiplicative inverses with respect to modulus f. As a verification, one can simply type **mod**$(e \cdot d, f)$ and see that they get 1.

In the finite math class, students are not expected to understand how the code for the **inverse** program works (it employs the Euclidean algorithm). However, they are expected to understand what a multiplicative inverse is and how it is verified. An important fact to note is that the enciphering exponent e must be chosen so that $\gcd(e, f) = 1$. If one chooses e where this requirement is not satisfied, the **inverse** routine will notify the user as demonstrated in Figure 4.

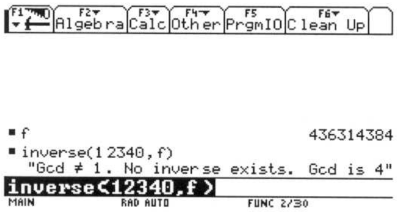

Figure 4

We next demonstrate how messages can be enciphered and deciphered using the TI–92 Plus. For example, suppose we desire to encrypt the message

high point university

with an RSA scheme created above. Figure 5 given below demonstrates how we encipher this message. To convert letters into numbers, we let $a = 01,\ b = 02,\ c = 03,\ \ldots,\ z = 26,\ [\ ] = 27$. Using the user–written function **digits** given in the appendix, we see that n is 9 digits long. Hence, we break our message into blocks of four letters (remember, each letter in general is translated into a two–digit number) to form "high", "poi", "nt u", "nive", "rsit", and "y" (the last block has the remaining number of leftover letters less than or equal to four). The user–written routine **tonumber** given in the appendix is designed to automatically perform this conversion for the specified number of letters. By entering,

tonumber("high point university", 4)→ptext

we translate this message in four letter blocks as "8090708", "27161509", "14202721", "14092205", "18190920", and "25" and store the result in a list as the variable *pt*. Using the user–written routine **expmod** (the code for this routine is given in the appendix), we encrypt each of these blocks as illustrated in Figure 5.

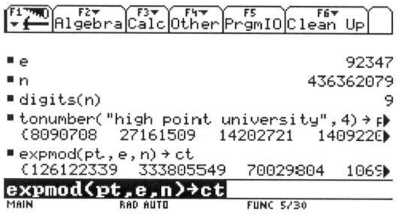

Figure 5

The previous calculation gives the cipher–text "126122339", "333805549", "70029804", "106900431", "37275443", and "318328073", which is stored as a list in the variable *ct*. To decipher, we use the deciphering exponent $d = 5094211$ found above to recover the message with **expmod**. Storing the result of **expmod** in the variable *pt*, we use the user–written function **toletter** in the appendix to recover the message. These calculations are illustrated in Figure 6.

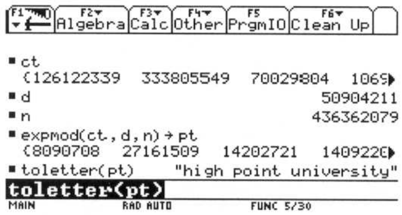

Figure 6

Recall that the security of the RSA cryptosystem is based on the inability of any intruder to factor the integer into a product of its prime factors p and q, which in turn prevents the intruder in determining the deciphering exponent d. For relatively small values of n, one can use the TI–92 Plus to demonstrate to students how to break and decipher a message encrypted using the RSA. For example, suppose an intruder intercepts the message encrypted as the integer blocks "39767444491", "13121743362", "12106110485", "11091994299", "34354345701", "48192903989" and "31880933212" using the recipients public key of $n = 89317065419$ and $e = 56909$. To decipher this message, we need to find the deciphering exponent d. To do this, the intruder factors n into its prime factors and determines $n = 123457 \cdot 723467$. The intruder then assigns $p = 123457$ and $q = 723467$ and finds $f = (p-1)(q-1) = 89316218496$. Next, the intruder determines that the inverse of $e \bmod f$ is the deciphering exponent $d = 18255921797$. The computations described can be seen in Figure 7.

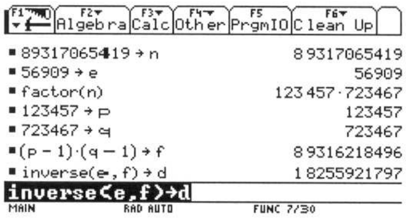

Figure 7

Finally, the intruder enters

$$\mathbf{expmod}\ (\{\mathbf{39767444491}, \mathbf{13121743362}, \mathbf{12106110485}, \mathbf{11091994299}, \mathbf{34354345701}, \mathbf{48192903989}, \mathbf{31880933212}\}, \mathbf{d}, \mathbf{n}) \rightarrow \mathbf{ptext}$$

which produces the integer blocks "318251620", "1507180116", "825270301", "1427020527", "501192527", "2015271205" and "11814" and stores them in the variable *ptext*.

By entering **toletter(ptext)**, we see that the message reads

cryptography can be easy to learn.

One can easily demonstrate how the security of the system can be increased. For example, one can quickly use the **nextprim** function to generate the two primes

$$p = 554545549391949639 \text{ and } q = 956985592342242422257$$

to form

$$n = p \cdot q = 426380581641242757566963409659471167152223.$$

However, it can quickly be seen that the TI–92 Plus is unable to factor this value of n in a reasonable amount of time.

An important note to make about the **expmod** program is that it employs a fast modular exponentiation algorithm similar to that found in [1]. A description of how this process works can be found in [2]. Although students in the finite mathematics class are not required to understand how the algorithm works, they do see the purpose of why this algorithm is beneficial. Performing the exponentiation before applying the mod requires working with extremely large integers, which quickly becomes computationally infeasible.

5. Conclusion. In this paper, we have demonstrated how the science of cryptography, specifically the RSA cryptosystem, can be integrated as part of a finite mathematics course. The use of the TI–92 Plus graphics calculator plays an important part in the teaching process. More details of how this concept is integrated into High Point's finite mathematics course can be obtained by requesting a copy of [3] from one of the authors.

The RSA Cryptosystem is currently one of the most common methods used for performing message concealment. To obtain more details on how and why the cryptosystem works, see [2].

Appendix: User–Written Codes. The code for the **nextprim**, **inverse**, and **expmod** routines is now given. Once these programs are downloaded into the TI–92 Plus, these routines can be accessed by typing their names on the calculator command line.

The **nextprim** function is specified by the form **nextprim(***number***)** and returns the smallest prime that is larger than the input parameter *number*. This function can be found in [4].

```
:nextprim(n)
:Func
:Loop
:n+1→n
:If isPrime(n)
:Return n
:EndLoop
:EndFunc
```

The **inverse** function is specified by the form **inverse(***number, modulus***)**, where *number* is the integer for which the multiplicative inverse is to be calculated with respect to the *modulus*.

```
:inverse(n,m)
:Func
:Local rm1,rm2,s,um1,um2,um3
:n→rm1
:m→rm2
:m→s
:1→um1
:0→um2
:While rm2>0
:mod(rm1,s)→rm2
:um1-um2*floor(rm1/s)→um3
:s→rm1
:rm2→s
:um2→um1
:um3→um2
:EndWhile
:If rm1≠1 Then
:Return "Gcd ≠ 1. No inverse exists. Gcd is "&string(rm1)
:Else
```

```
:If um1<0
:mod(um1,m)→um1
:Return um1
:EndIf
:EndFunc
```

The **expmod** function is specified by the form **expmod(***base, exponent, modulus***)**, where *base* is the exponential base raised to the *exponent* power and reduced to the integer remainder upon division by the *modulus*. The first parameter *base* can either be a single integer or a list of integers enclosed in curly brackets { } separated by commas.

```
:expmod(b,e,m)
:Func
:Local bc,y
:b→bc
:1→y
:While e>0
:If mod(e,2)=1
:mod(bc*y,m)→y
:floor(e/2)→e
:mod(bc∧2,m)→bc
:EndWhile
:Return y
:EndFunc
```

The **digits** function is specified by the form **digits(***number***)**, where *number* is the non-negative integer for which the number of digits is to be counted.

```
:digits(n)
:Func
:Local ct
:string→ct
:Return dim(ct)
:EndFunc
```

The function **tonumber** is specified by the form **tonumber(***message, numletter***)**, where *message* is the message string specified in lower case letters enclosed in quotation marks that will be converted to its alphabet numerical representation in *numletter* blocks.

```
:tonumber(mess,nl)
:Func
:Local sl, cn, sn, ii, jj, alphabet, numl
:“abcdefghijklmnopqrstuvwxyz” → alphabet
:dim(mess)→sl
:{}→numl
:0→ii
:While ii<sl
:0→cn
:0→jj
:While jj<nl and ii+jj<sl
:1+jj→jj
:inString(alphabet,mid(mess,ii+jj,1))→sn
:100*cn+sn→cn
:EndWhile
:augment(numl,{cn})→numl
:ii+nl→ii
:EndWhile
:If dim(numl)=1 Then
:Return numl[1]
:Else
:Return numl
:EndIf
:EndFunc
```

The function **toletter** converts a number or list of numbers into its alphabet representation. It is specified by the form **toletter(***number***)**, where the parameter *number* is the single number or list of numbers enclosed in curly brackets { } to be converted.

```
:toletter(mnum)
:Func
:Local ii,jj,cs,cn,sl,sans,bl,numl,bstr, 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”, “ ” }→alphabet
:””→sans
:{mnum}→numl
:If dim(numl)≠1
:mnum→numl
:dim(numl)→bl
```

```
:For jj,1,bl
:numl[jj]→cn
:floor(log(cn)/2.))+1→sl
""→bstr
:For ii,1,sl
:cn/100→cn
:alphabet[fpart(cn)*100]→cs
:cs&bstr→bstr
:floor(cn)→cn
:EndFor
:sans&bstr→sans
:EndFor
:Return sans
:EndFunc
```

References

1. T. Feil, "RSA Encryption," *MapleTech,* 2 (1996), 50–52.
2. R. Klima, N. Sigmon, and E. Stitzinger, *Applications of Abstract Algebra With Maple,* CRC Press, Llc., Boca Raton, FL, 2000.
3. N. Sigmon, "Cryptography: Chapter for Finite Mathematics," Mathematics Department, High Point University.
4. *Texas Instruments TI–89 TI–92 Plus Guidebook for Advanced Mathematics Software Version 2.0,* Texas Instruments, 1999.

Mathematics Subject Classification (2000): 00-01, 97-04

Robert T. Harger
Department of Mathematics
High Point University
High Point, NC 27262
email: rharger@highpoint.edu

Neil P. Sigmon
Department of Mathematics
High Point University
High Point, NC 27262
email: nsigmon@highpoint.edu