Finding Inverses in a Finite Field

New Mexico

Supercomputing Challenge

Final Report

April 5, 2006

Team Number 54

Manzano High School

Team Members

Kristin Cordwell

Chen Zhao

Teacher

Mr. Stephen Schum

Project Mentor

Dr. William Cordwell

Table of Contents

Executive Summary....page ii
Introduction....page 1
The Extended Euclidean Algorithm....page 3
Extended Euclidean Algorithm - Examples....page 4
Euclidean Algorithm Flowchart....page 6
Extended Euclidean Algorithm Flowchart....page 7
Lagrange's Theorem....page 8
Lagrange's Theorem - Examples....page 9
Speedup Techniques for Lagrange Code....page 10
Lagrange's Theorem Flowchart....page 13
More Advanced and Faster Speedup Techniques....page 14
A Computer Implementation Problem and Its Solution....page 16
Results....page 17
    Figure 1, Speeds of LGT Programs....page 17
    Figure 2, Speeds of EEA Programs....page 18
Conclusions....page 19
    Figure 3, Lagrange vs. EEA (Fastest Methods)....page 19
Applications....page 20
Bibliography....page 20
Appendix A – Groups....page 21
Appendix B – Fields....page 22
Appendix C – Code
    Extended Euclidean Algorithm, Bit Arrays....page 24
    Extended Euclidean Algorithm, Byte Arrays....page 35
    Extended Euclidean Algorithm, Byte Arrays & Reduction Table....page 43
    Lagrange's Theorem (Algorithm), Russian Peasant....page 53
    Lagrange's Theorem (Algorithm), Itoh's Technique....page 60
    Lagrange's Theorem (Algorithm), Itoh's & Byte Arrays....page 71
    Random Polynomial Generator....page 81
Appendix D – Results Data Tables....page 83
Sample Result Verification Using Mathematica....page 89

Executive Summary

To find multiplicative inverses in a finite field, one may use one of two methods: either the Extended Euclidean Algorithm (EEA) or Lagrange's Theorem. The EEA is a recursive method that generates successively smaller remainders, down to 1 (if an inverse exists), then uses the coefficients that were obtained to build the inverse. Lagrange's Theorem says that taking an element to a large power will give the inverse. Which method is more efficient?

We specialized to finite fields in the class GF(2^n), which occur most naturally in computer systems, and we chose GF(2^{155}) as a representative case, as it is of a size to be of "real-world" interest, for example, in Elliptic Curve Cryptography. Also, since a polynomial of degree 154 is much too large to store or manipulate using even 64-bit words, it forced us to develop the machinery to work with large polynomials stored as arrays; our code should easily generalize to polynomials of an arbitrary size.

The bulk of our work was in applying known methods and developing some of our own to speed up both algorithms to an optimal level. At first, we stored the coefficients of the polynomials as one bit per array element, as this is straightforward and lends itself to correct implementation. For the Lagrange Method, which involved raising a polynomial to the power $2^{155} - 2$ (yes, that's actually the exponent!), we first used the Russian peasant method outlined in Knuth [2]. After more research, we found a method to greatly reduce the number of time-intensive polynomial multiplications involved [6]. This provided a dramatic increase in performance.

We decided to compact our coefficients by storing the maximum number in each array element. Here, we decided to use bytes, rather than, say, 32-bit words, because we realized that we could precompute and use many lookup tables if they were indexed by pairs of bytes. Applying these ideas in these ways is our most significant original achievement. These techniques yielded an additional factor of approximately fifteen for each of the two methods.

Overall, we found that the Extended Euclidean Algorithm is a slightly faster method of finding inverses in GF(2^{155}) than is the method of Lagrange's Theorem.

Introduction

To find multiplicative inverses in a finite field, one may use two general methods: the Extended Euclidean Algorithm, or Lagrange's Theorem.

Given two relatively prime integers, p and q, the Extended Euclidean Algorithm may be used to find two other integers, a and b, such that $a \cdot p + b \cdot q = 1$. If we consider this equation modulo p (dividing each individual term by p and retaining the remainder), we obtain the equation $0 + b \cdot q = 1 \bmod p$. The $a \cdot p$ term is thus cancelled and we can see that the multiplicative inverse of q mod p is b, because b times q = 1.

Alternately, Lagrange's Theorem from Group Theory says that, for a finite group G, any element raised to the order of the group, O(G), equals 1 (see Appendix A). This means that if an element is raised to one less power, O(G) - 1, it will give the multiplicative inverse of the element. Since the numbers that are relatively prime to p form a multiplicative group, this provides an alternate method for finding a multiplicative inverse mod p.

Both of these methods may be extended to general finite fields (see Appendix B). In particular, $GF(2^n)$ is the finite field of polynomials with coefficients in Z_2 (the integers modulo 2), with each polynomial being taken modulo an irreducible polynomial of degree n. There are 2^n elements in the field, and, discarding the zero element, there are 2^n-1 elements in the multiplicative group $GF(2^n)^*$, so we may find the inverse of an element by raising it to the 2^n-2 power. Alternately, we may use the Extended Euclidean Algorithm, starting with the given field element (polynomial) and the irreducible polynomial to find the inverse.

Finite fields, especially $GF(2^n)$, are used extensively in such fields as cryptography and error correction codes. For example, the Advanced Encryption Standard (AES) calculates the inverse of elements in $GF(2^8)$ in each round of the encryption. Elliptic Curve Cryptography also makes use of finite field inverses. Reed-Solomon and other BCH Error Correcting Codes, which are extensively used in communications and data transmissions, also use inverses of elements in $GF(2^n)$. It is therefore useful to be able to efficiently calculate the inverse of these field elements.

In our project, we wrote code to implement both of these algorithms for finding inverses of elements of $GF(2^n)$. We chose $n = 155$ as a specific case, because it is a size that is of interest in elliptic curve cryptography, and because the field elements are too large to be handled using single16-bit or 32-bit words, but rather must be stored and manipulated as arrays. The methods that we used, then, are applicable to any general field $GF(2^n)$.

We discovered or developed and used various speedup techniques to improve the efficiency of each algorithm, and measured the run-times of each against hundreds or thousands of random polynomials. The bulk of our effort was in applying these various speedup techniques. These are described in detail, below.

The Extended Euclidean Algorithm

The Euclidean Algorithm is a method developed by the Greek mathematician Euclid that finds the greatest common divisor of two positive integers. We begin with two numbers, say a and b, where a is greater than b. We divide a by b and get a remainder r. We then divide b by r, to get a new remainder t. We continue in this manner until the remainder is either one or zero. If the remainder is one, then the two numbers are relatively prime, so their greatest common divisor is one. However, if the remainder is zero, the greatest common divisor would be the remainder before the zero.

The Extended Euclidean Algorithm takes two relatively prime integers, x and y, where x is greater than y, and finds the inverse of y mod x. It works the same way as the Euclidean Algorithm does; however, we keep track of the quotients when we divide x by y, and then we use them to find the multiplicative inverse of y mod x (see examples below).

We may also use the Extended Euclidean Algorithm to find the multiplicative inverse of polynomials in GF(2^n). We do this by starting with the given polynomial and taking it mod the irreducible polynomial. We go through the steps of the algorithm and thus obtain the inverse.

Extended Euclidean Algorithm – Examples

Example # 1 (Integers)

Find the inverse of 8 mod 19:

$19 = \underline{2} \cdot 8 + 3$	8 goes into 19 twice, with a remainder of 3
$8 = \underline{2} \cdot 3 + 2$	take the remainder 3 and divide it into 8; it goes in twice with r = 2
$3 = \underline{1} \cdot 2 + 1$	take the remainder 2 and divide it into 3, it goes in once with r = 1 (when r = 1, we are done with this part). The quotients are underlined.

	1	0		start this with two rows, 1 0 and, underneath it, 0 1
(−)	0	1	(·2)	multiply the lowest row by our first quotient, two, and subtract it from the previous row to get:
(−)	1	−2	(·2)	multiply this by our second quotient, two, and subtract it from the previous row to get:
(−)	−2	5	(·1)	multiply this by our third quotient, one, and subtract it from the previous row to get:
	3	−7		

This means that

$3 \cdot 19 - 7 \cdot 8 = (+/-)\ 1$

We take this mod 19, to obtain

$0 - 7 \cdot 8 = 1$

Thus, the inverse of 8 is (-7) mod 19, which is the same as 12 mod 19. We double check: $8 \cdot 12 = 96$, and 96 divided by 19 is 5, with a remainder of 1.

Example # 2 (Polynomials)

Find the inverse of x^2 in the field GF(2^3):

In order to find this inverse, we take x^2 mod an irreducible polynomial; $x^3 + x + 1$

Also, remember that in mod 2, +1 is the same as -1.

$x^3 + x + 1 = \underline{x} \cdot x^2 + (x + 1)$

$x^2 = \underline{x}(x + 1) + x$

$x + 1 = \underline{1}(x) + 1$ where the quotients are underlined

Recall that subtraction mod 2 is the same as addition mod 2. We then have

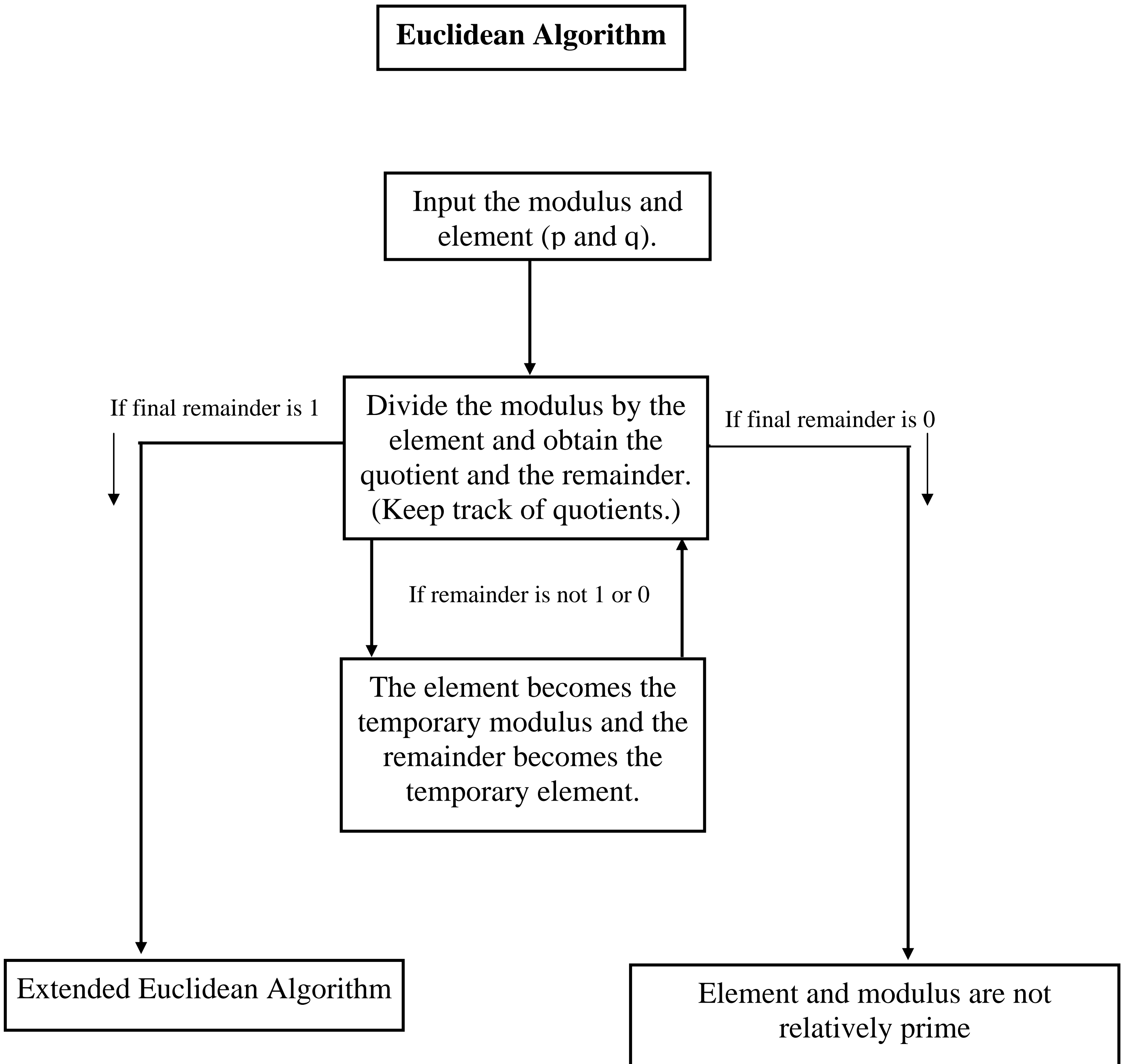
1	0	
0	1	multiplying by x and subtracting from the previous row,
1	-x (= x)	multiplying this row by x and subtracting from the previous
x	$x^2 + 1$	multiplying by 1 and subtracting from the previous
x + 1	$x^2 + x + 1$	

Thus, $(x + 1)(x^3 + x + 1) - (+)(x^2 + x + 1)(x^2) = 1$

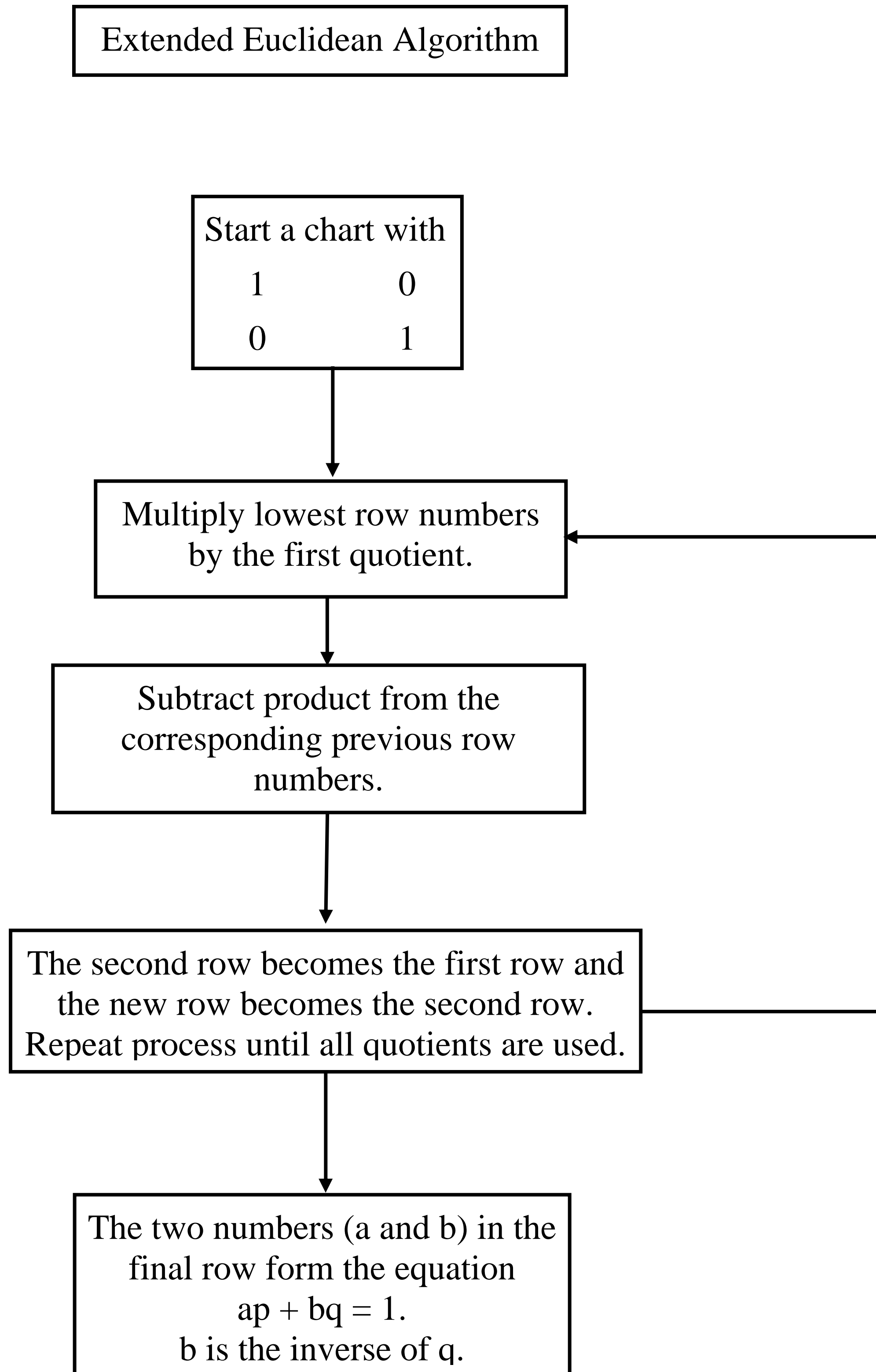
When we take this mod $(x^3 + x + 1)$, we get $0 + (x^2 + x + 1)(x^2) = 1$

Thus, the inverse of x^2 is $(x^2 + x + 1)$.

Euclidean Algorithm Code Flowchart



Extended Euclidean Algorithm Code Flowchart



Lagrange's Theorem

Lagrange's Theorem says that, in a finite group G, if we raise an element of the group to the order of the group (the number of elements in G), we get the identity. This implies that by raising an element to one less than the order of the group, we will obtain the multiplicative inverse of the element.

If we apply Lagrange's Theorem to integers, we note that Z_p* (the integers mod p, not counting 0) is a multiplicative group when p is prime, so we can carry out the process.

Alternately, we can apply Lagrange's theorem to the finite field GF(2^n), where we have polynomials with coefficients modulo two. We take our polynomial that we wish to find the inverse of modulo an irreducible polynomial of degree n. Since there are $2^n - 1$ elements in the multiplicative group (0 is not in the multiplicative group), we may find the inverse of a polynomial by raising it to the $2^n - 2$ power.

Lagrange's Theorem – Examples

Example # 1 (Integers)

Find the inverse of 8 mod 19:

We take 8 mod 19 and raise it to the power of the order of the group minus 1:

(19 – 1 – 1) = 17

8^{17} mod 19 = 12

Thus, 12 is the inverse of 8 (mod 19).

Example # 2 (Polynomials)

Find the inverse of x^2 in the field GF(2^3):

The order of the field is 2^3 elements, or eight elements.

However, this includes zero, so we take away one element, leaving us with seven elements in the multiplicative group GF(2^3)*.

We choose the irreducible polynomial $x^3 + x + 1$ as our modulus.

Now we subtract one from the order of the group, to get six, so

$(x^2)^6 = x^{12}$ is the inverse of x^2 mod $x^3 + x + 1$ in the multiplicative group, and also in the original field.

Since the modulus (essentially, the zero element) of the field is $x^3 + x + 1 = 0$, we may raise it to any power and it will still be the zero element. This also says that $x^3 = x + 1$.

Raising it to the fourth power gives us

$x^{12} = (x+1)^4 = x^4$ + [even number] x^3 +[even number] x^2 + [even number] + 1

Since we are GF(2^3), the even numbers equal zero, so we get $x^{12} = x^4 + 1$, which is the same as $(x+1)x+ 1$, which equals $x^2 + x + 1$.

Thus, the inverse of x^2 in the field GF(2^3) is $x^2 + x + 1$.

SpeedupTechniques for Lagrange Code

Intermediate Reductions

As we go through Lagrange's Theorem, our polynomial that we are raising to the power will get very large. Large polynomials take even longer to multiply or square. In order to avoid storing and manipulating such large polynomials, and thus slowing down our run time, we reduce the intermediate values as we go. Because of the mathematics of modular reduction, which applies to polynomials as well as integers, the answer will be the same.

Using a Trinomial for the Irreducible Polynomial

While not every finite field $GF(2^n)$ has an irreducible trinomial as its modulus, many do, and all irreducible polynomials of degree n give equivalent fields. By using a trinomial, (or at least a polynomial with only a few terms), we may do modular reduction more efficiently.

Squaring Polynomials the Fast Way

In our programs, we make use of a trick for squaring polynomials in order to expedite our run times.
For example, if we want to square $x^3 + x + 1$, we write it as:
$1 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x^1 + 1 \cdot x^0 = 1011$.
Then, to square it, we put a zero in front of each digit (or between each original pair) to get:
$01000101 = 1 \cdot x^6 + 0 \cdot x^5 + 0 \cdot x^4 + 0 * x^3 + 1 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0$
We check by doing out the squaring longhand:
$(x^3 + x + 1)^2 = x^6 + 2x^4 + 2x^3 + x^2 + 2x + 1 = x^6 + x^2 + 1 \pmod 2$

This means that, rather than actually multiplying a polynomial by itself to square it, we can just expand it in this way.

Russian Peasant Method versus Itoh's Technique

The modulus (irreducible polynomial) that we are using is $p(x) = x^{155} + x^{62} + 1$. The field $GF(2^{155})$ has 2^{155} elements, and the corresponding multiplicative group, $GF(2^{155})^*$, has $2^{155} - 1$ elements. To find the inverse, then, we need to raise an element to the $2^{155} - 2$ power. Notice that, in binary

10000000…00 (155 zeros)
-10
———————
11111111…10 (154 ones, one final zero)

By the Russian Peasant Method, Lagrange's Theorem thus has 154 squarings and 153 multiplications, as follows. From Knuth [2], we do these by the following method: list the exponent in binary, and follow it from the high bit down to the lowest bit. Start by "writing" the number; this corresponds to the high bit being a 1. Shift down one bit in the exponent. Every time we encounter a one, we square and multiply the stored number, and every time we have a zero, we simply square the stored number. Then shift down one more bit and repeat.

For example:

Say we want:	a^{13}	1101 (equals 13 in binary)	
We start with:	a^1	$\underline{1}101$	
Shift down one bit:		$1\underline{1}01$	
Square and multiply:	$a^2 \cdot a = a^3$	$1\underline{1}01$	then shift down one bit
Square:	$(a^3)^2 = a^6$	$11\underline{0}1$	then shift down one bit
Square and multiply:	$(a^6)^2 \cdot a = a^{13}$	$110\underline{1}$	

For $2^{155} - 2$ = 111111111...1111110 (in binary), we thus have 153 squares and multiplies, and 1 additional squaring.

This is a lot of “multiplies”. In the third technique, above, we noticed that squaring is very quick; multiplying, on the other hand, is much more computationally intensive. We’d like to minimize the number of multiplications.

In 1986, Itoh, Teechai, and Tsujii noticed that[6]:

$2^{155} - 2 = 2 \cdot (2^{77} - 1) \cdot (2^{77} + 1)$

$2^{77} - 1 = 2 \cdot (2^{19} - 1) \cdot (2^{19} + 1) \cdot (2^{38} + 1) + 1$

$2^{19} - 1 = 2 \cdot (2^{9} - 1) \cdot (2^{9} + 1) + 1$

$2^{9} - 1 = 2 \cdot (2 + 1) \cdot (2^{2} + 1) \cdot (2^{4} + 1) + 1$

Since we are dealing with the exponent, each “+1” indicates a multiplication, while twos would involve squaring..

For example, if we wish to raise an element to the $2^9 - 1$ power,

$x^{2(2+1)} = (x^2)^2 \cdot x^2 = y$

$y^{(2 \cdot 2+1)} = (y^2)^2 \cdot y = z$

$z^{(2 \cdot 2 \cdot 2 \cdot 2+1)} = (z^2)^4 \cdot z = t$

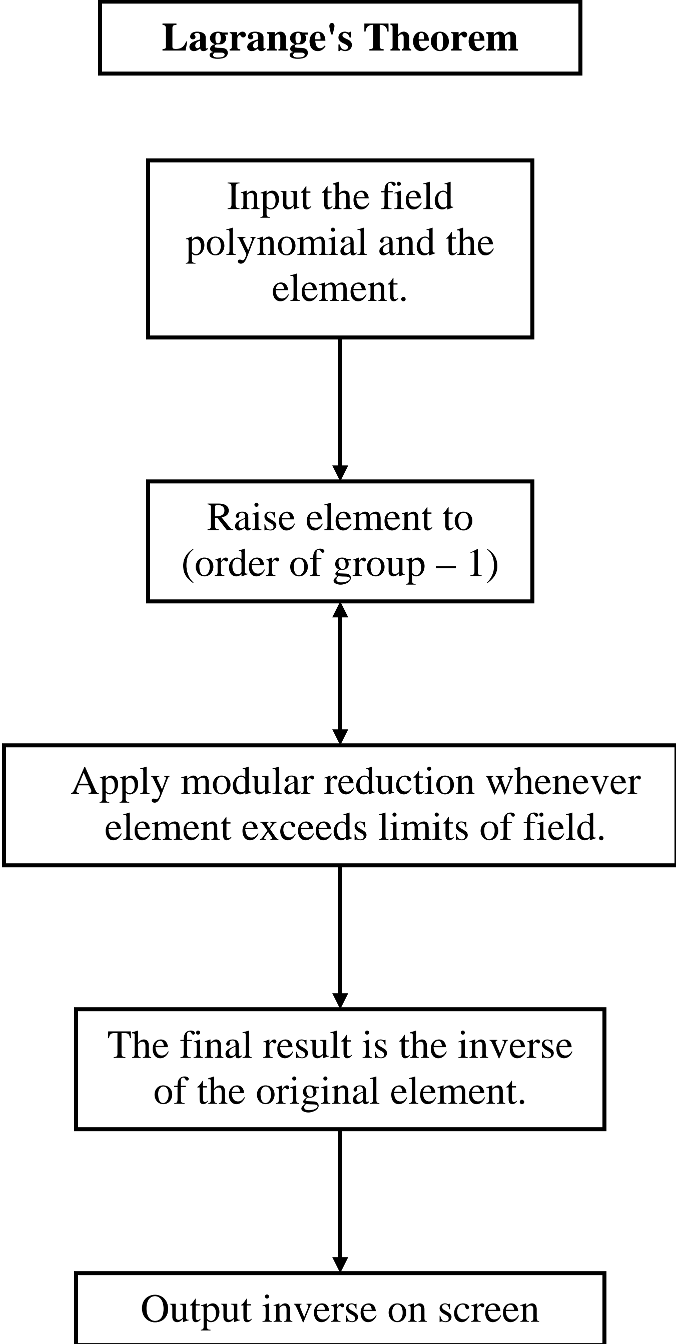
and, finally, multiplying again by x for the final “1”,

$t \cdot x$. = x raised to the $2^9 - 1$ power.

Thus, $x^{2(2+1)(2 \cdot 2+1)(2 \cdot 2 \cdot 2 \cdot 2+1)}$ has only four multiplications, as well as many squarings.

So $2^{155} - 2$, by Itoh’s technique, has only 10 multiplications and 154 squarings, versus 153 multiplications and 154 squarings by Knuth’s technique. Itoh’s technique is thus much faster, so we use it in our code rather than Knuth’s technique.

Lagrange’s Theorem Code Flowchart



More Advanced and Faster Speedup Techniques

Storing the Coefficients Packed Together in a Larger Data Size Unit

Up to this point, we had been representing each polynomial as an array of ones or zeros. The index of the array corresponded to the power of the x involved, and the value of one or zero corresponded to the value of its coefficient. For simplicity, and to get the program working, we stored a single coefficient in a single array element. This meant, for example, that to multiply two polynomials of degree 154 (with 155 coefficients, including the one for x^0) required $155^2 = 24025$ separate comparisons. However, since a computer is designed to work on data sizes of bytes (8 bits), 16-bit words, or 32-bit words, we thought that we could get much better performance by storing a group of coefficients together in a larger data size.

It was tempting to try to use the largest size available, but we realized that by using bytes, we could precompute many values and store them in tables. A table involving pairs of bytes would need to have a size of $256 \times 256 = 64$ k units of storage, which is a reasonable size to store in memory, while using pairs of 16-bit unsigned integers would give $2^{32} = 4$ G units of storage, which seemed less feasible for a PC, especially if several tables were involved. Clearly, pairs of words would be infeasible for even the largest current memories.

Storing a polynomial of degree 154 (155 bits) then requires 19 bytes of storage.

Table for Multiplication of Bytes

Since we are really multiplying polynomials over the base field Z_2 , we can't use the regular integer multiplication that is already programmed into the microprocessor, so we needed to build our own table. Since two 8-bit "numbers" multiplied together give a 16-bit answer, we made an array of double bytes, a high byte and a low byte, indexed by the two bytes that were input.

Multiplication of two polynomials now involves $19 \times 19 = 361$ multiplication table lookups, along with keeping track of the output byte positions and the attendant xorings.

This is far less than the single coefficient storage multiplication described above.

Table for Squaring

When each byte is squared, because of the peculiarity of the polynomials over Z_2, a byte such as hgfedcba will expand to 0h0g0f0e 0d0c0b0a. Such a table, indexed by a single byte, gives a double byte output that is easy to compute. The entire polynomial is easy to square, since the 5th byte of the original polynomial, for example, gets squared and put into the 10th and 11th bytes of the squared polynomial.

Tables for Modular Reduction

After squaring or multiplication, the resulting polynomial typically is much "bigger" (has a higher degree, or largest power) than the modulus polynomial, $x^{155} + x^{62} + 1$. Since, mathematically, this is equivalent to saying that $x^{155} = x^{62} + 1$ (recall that minus equals plus in Z_2), to reduce, we replace every x^{155} with $x^{62} + 1$. This means that, for example, $x^{300} = x^{155} \cdot x^{145} = (x^{62} + 1) \cdot x^{145} = x^{207} + x^{145}$. The x^{207} in turn becomes $(x^{62} + 1) \cdot x^{52}$. In terms of computer programming, this means that a "1" bit corresponding to a power greater than or equal to 155 gets xored with the bits corresponding to the bits 93 (which is 155 – 62) and 155 lower in the polynomial representation. A little thought shows that we can do this process an entire byte at a time, thus, the byte with index 22 (starting with index 0) will correspond to powers of x between 176 and 183 (inclusive). This whole byte must be xored with the powers of x shifted down by 93 and 155, respectively. That is, with the powers corresponding to 83 to 90, and to 21 to 28. The bits corresponding to these powers, unfortunately, do not fall on byte boundaries, so we must split them into two bytes and xor the pieces with the bytes containing the respective powers. Also, the splits corresponding to shifting down by 93 and 155 occur in different places. What we did was create two tables of byte pairs, indexed by the input byte that we could then xor in the appropriate places. For byte index 19, which contains the bit corresponding powers 152 to 159 (containing 155), we built separate tables, since we would be dealing with only the upper part of that byte.

A Computer Implementation Problem and Its Solution

When we ran our code, we looped over multiple iterations of a polynomial, because any single computation was too fast to be statistically meaningful. We also put in an outer loop, to be able to take an average of the multiple iteration times. However, on examining the output, we noticed that the values would be close for a while, then they would drop fairly dramatically, then (sometimes) they would come back up in value.

At first, we thought that this might be because of computer caching, which is when a computer runs particular instructions many times, and it places those instructions (or data) in special storage near the micro-processor, which makes them much quicker to access.

Because we didn't want this caching to give us incorrect data (we're trying to find the time to calculate an average inverse), we built a table of (quasi-) random polynomials (we used 50 for the single bit per array element code and 500 to 1000 for the eight bits per array element code, but various other values should give the same benefit), and, every time that we computed an inverse, we switched the input polynomial.

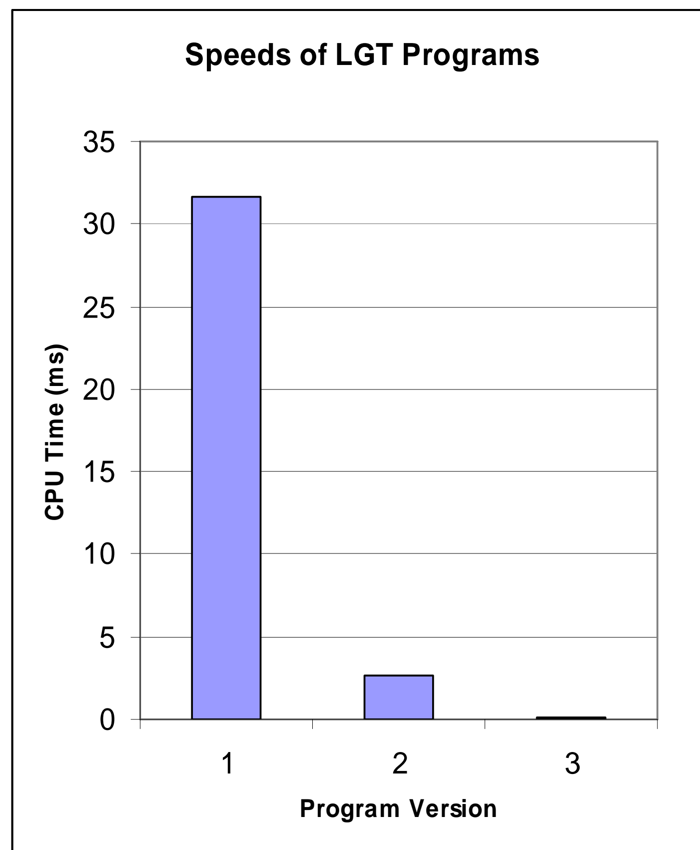
This did not solve the overall problem, although it certainly made better sense, statistically, to use multiple random polynomials.

Later, we noticed the same effect, and had Task Manager running at the same time. It was then immediately apparent that a program called DSSAGENT.EXE was activating at intervals, and that these times coincided with the times when the program speed slowed significantly. Setting the priority of our program to the highest level and that of DSSAGENT.EXE to the lowest level helped some, but the latter program would still activate and take close to 50 percent of the CPU time. Checking on the Internet revealed that DSSAGENT.EXE seemed to be associated with adware / spyware, and that most people recommended deleting it. After we did this, the run times smoothed out dramatically. As a precaution, we still shut down all other user programs except our polynomial program, and also set its priority to the highest level.

Results

For our final data, we ran six programs; three each for the Extended Euclidean Algorithm and Lagrange's Theorem. First we ran the original EEA code. Then we executed the EEA code which was stored in bytes rather than bits, and had a byte multiplication table for building the inverse. Finally, we ran our latest EEA code, which was stored in bytes with a byte multiplication table as well as an extra lookup table for the shifting and xoring process. Similarly, we began the Lagrange's Theorem data by executing the original program, in which we used the Russian Peasant Method and stored in bits. Next we ran Lagrange's Theorem code which used Itoh's method rather than the Russian Peasant Method, and finally we executed our latest Lagrange's Theorem Code, with Itoh's method and byte arrays as well as lookup tables [see Appendix D for data tables].

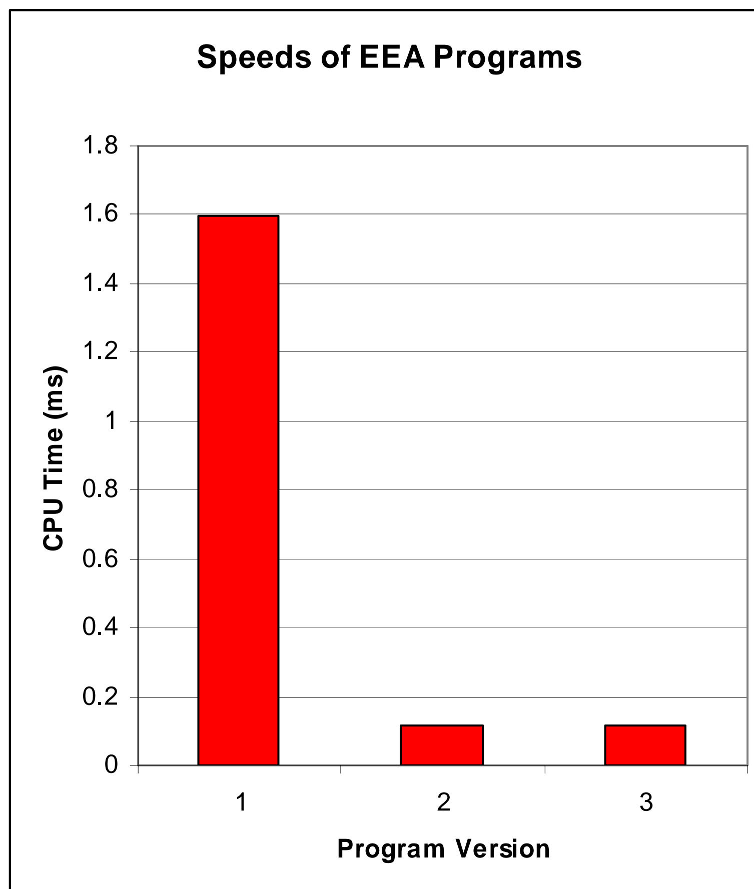
To begin with, we compared the improvements within a single inverse algorithm, to find how much the speedup techniques actually helped our run time. We found that, for



(Fig. 1)

Lagrange's Theorem, the improvements were quite startling, with Itoh's technique being nearly 12 times faster than the Russian Peasant Method, and byte storage / lookup tables improving upon that time by another factor of 17.5. Thus, our final time for Lagrange's Theorem of 151 microseconds per polynomial was 209 times faster than our original time.

Our results for the EEA were not quite as we expected. The byte storage and multiplication lookup table made a substantial difference, improving performance by a factor of 13.7. However, using an extra lookup table for byte shifting actually slowed the process by 1/10 microsecond. Though not statistically significant, this was unexpected, as we were hoping that it would speed things up. As it turns out, in the EEA, typically, there is only a difference of 2^+ bits between the lengths of the polynomials during the reduction process. To the computer, this means that looking up values a couple of times and xoring twice is no faster than directly shifting and xoring two times each. Our fastest EEA time was 116.5 microseconds.

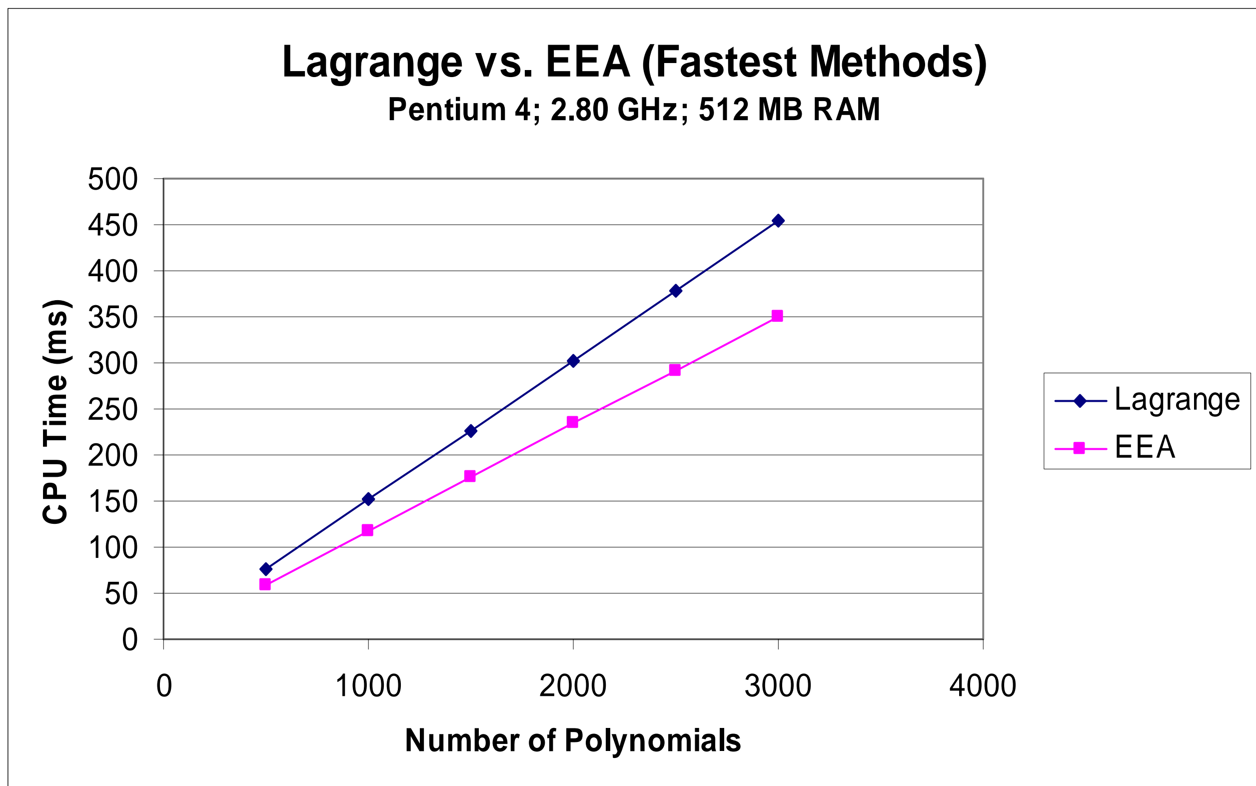


(Fig. 2)

Conclusions

There were three techniques that yielded major improvements in performance: using Itoh's Technique to reduce the number of multiplications increased the speed by a factor of twelve. The second technique, storing eight coefficients per array element, taken in conjunction with the third, lookup tables for multiplication and modular reduction, improved the performance of both algorithms by, roughly, another factor of fifteen.

Overall, the Extended Euclidean Algorithm was faster, with a run time of 116.5 microseconds to invert one polynomial, compared with a run time of 151 microseconds per polynomial for the Lagrange Theorem method. This may be seen in Figure 3, where the slope (run time per polynomial) is slightly less for EEA than the slope for the Lagrange method implementation.



(Fig. 3)

Applications

Finite fields, especially $GF(2^n)$, are used extensively in such fields as cryptography and error correction codes. For example, the Advanced Encryption Standard (AES) calculates the inverse of elements in $GF(2^8)$ in each round of the encryption. Elliptic Curve Cryptography also makes use of finite field inverses, with sizes similar to those used in our project, $GF(2^{155})$. Reed-Solomon and other BCH Error Correcting Codes, which are extensively used in communications and data transmissions, also use inverses of elements in $GF(2^n)$.

Bibliography

[1] Hankerson, Hernandez, and Menezes,
"Software Implementation of Elliptic Curve Cryptography over Binary Fields"
<u>CHES 2000 Proceedings</u>, Springer-Verlag, Berlin, 2000

[2] Knuth, Donald, "The Art of Computer Programming, vol. 2, Seminumerical Algorithms"; Addison Wesley Longman, Berkeley, CA, 1998

[3] Lidl and Niederreiter, "Finite Fields";
Cambridge University Press, New York, 1997

[4] McEliece, Robert J., "Finite Fields for Computer Scientists and Engineers";
Kluwer Academic Publishers, Boston, 1987

[5] Menezes, Alfred J., Editor, "Applications of Finite Fields";
Kluwer Academic Publishers, Boston, 1993

[6] Menezes, "Elliptic Curve Public Key Cryptosystems";
Kluwer Academic Publishers, Boston, 1993

[7] Rotman, Joseph J., "A First Course in Abstract Algebra";
Prentice-Hall, Upper Saddle River, NJ, 2000

[8] Schroeppel, Orman, O'Malley,
"Fast Key Exchange with Elliptic Curve Systems"
University of Arizona, 1995

Appendix A – Groups

A group is a set of elements G and an operation "*" with the following properties:

If g and h are elements of G, then g * h is in G (closure)

There exists an identity element e in G such that, for every g in G, e*g = g

For every element g in G, there exists an inverse element g^{-1} such that g^{-1} *g = e

If g, h, and k are in G, then g· (h·k) = (g*h)*k (associativity)

It is important to notice that the operation (*) could be regular multiplication, but it could also be addition, or any other operation that satisfies the group properties.

Some familiar groups are:

the integers (…−2, −1, 0, 1, 2, …) under addition

the non-zero rational numbers under multiplication

Z_n : the integers modulo n (the remainder after dividing by n) under addition

Z_p* : the non-zero integers modulo a prime p under multiplication

we leave out zero because it has no multiplicative inverse; that is, we cannot divide by zero

also, the modulus must be a prime, because otherwise not every element has a multiplicative inverse

For a group G with a finite number of elements, the order of the group is defined to be the number of elements, written as O(G).

Appendix B – Fields

Introduction:

A field is a set of elements and two operations, called addition and multiplication, which is:

- a commutative group under addition
- a commutative group under multiplication (excluding 0 for having an inverse)
- is distributive

Some well-known fields are:

- rational numbers
- real numbers
- complex numbers
- Z_p (the integers modulo a prime, p)

Of these four examples, the first three have an infinite number of elements, and the last one has a finite number of elements, p. This last is thus an example of a finite field.

As before, the order of a finite field is the number of elements in it.

General Finite Fields:

Finite fields are based on an underlying finite field, Z_p, where p is a prime. From this, we can build up a more general field, called a Galois Field (GF), of order p^n, where n is a positive integer. We do this by first making an n-dimensional vector space. For example, we can choose n = 8, over the underlying field Z_2. Thus, we can represent any element of $GF(2^8)$ by its coordinates, for example, (1, 0, 0, 1, 0, 1, 1, 1). What makes this a field and not just a vector space is the fact that we define a multiplication operation. Usually this is done as follows:

Consider the equivalent representation in terms of a polynomial. Each coordinate corresponds to a different power of x. For example, (1, 0, 0, 1, 0, 1, 1, 1) would correspond to $g(x) = 1*x^7 + 0*x^6 + 0*x^5 + 1*x^4 + 0*x^3 + 1*x^2 + 1*x + 1$. We then

pick an irreducible polynomial, which is the equivalent of a prime number for integers. This polynomial is irreducible because it has no non-trivial polynomial factors. For example, for $GF(2^8)$, we may pick $x^8 + x^4 + x^3 + x + 1$ as our irreducible polynomial. Now, for example, we can multiply g(x) by itself to get $g(x)^2 = 1{\cdot}x^{14} + 1{\cdot}x^8 + 1{\cdot}x^4 + 1{\cdot}x^2 + 1$, where we have left out all the zero terms. Notice that, when we square a polynomial, we just square the individual terms; this is because the cross-terms all have coefficient 2, which is 0 in Z_2. We then need to reduce by the irreducible polynomial, which means we find the remainder after doing polynomial long division. The result is $x^7 + x^4 + x^2$, which we could also write in coordinate form as (1, 0, 0, 1, 0, 1, 0, 0).

In defining the finite field, typically, there are several irreducible polynomials of the necessary degree; we can use any one of them, because they all give the same basic field.

Note that the multiplicative group associated with a finite field is all of the elements except the zero element, so the group has one fewer element than the field. We write the multiplicative group as $GF(p^n)^*$.

Appendix C – Code

Extended Euclidean Algorithm, Bit Arrays

```
/*
   Extended Euclidean Algorithm for Polynomials
   one bit per array element
   Chen Zhao, Kristin Cordwell
   03-15-06
*/

#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>
#include <ctime>

using namespace std;

const unsigned int MAX=201;                    //  MAX-1 is the highest degree of any
polynomial used

typedef unsigned char coef;
coef ch0 = '0';
coef ch1 = '1';

char infilename[] = "poly_list.txt";           //keep this name
const int num_polynomials = 50;                      //up to 50 allowed
int kloop=6*num_polynomials;
int nloop=50;
//coef   poly[num_polynomials][MAX];                         //work with this many
polynomials
//int   degree[num_polynomials];

typedef struct
{
  coef bit[MAX];
  int  deg;
} polytype;

polytype poly[num_polynomials];

void intro();
void poly_read(int, polytype*);
void poly_write(int, coef*, int, coef*, int, coef*, int, coef*);
void poly_factor(int, coef*, int, coef*, int&, coef*, int&, coef*);
void poly_multiply(int, coef*, int, coef*, int&, coef*);
```

```
void poly_add(int, coef*, int, coef*, int&, coef*);
void poly_copy(int, coef*, int&, coef*);
void polytype_copy(polytype, int&, coef*);
int  poly_degree(int, coef*);
void poly_output(int, coef*);
void poly_ofile(int, coef*, ofstream&);
void poly_init(coef*);

coef a[MAX] = {'1','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','1','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','0',
        '0','0','0','0','0','0','0','0','0','1'};
int  da = 155;
//a is the irreducible polynomial that generates the field
//a = x^155 + x^62 + 1

//=============================================================
============
int main()
{
  coef at[MAX], bt[MAX], q[MAX], r[MAX];
  //  arrays
  coef x1[MAX], y1[MAX], x2[MAX], y2[MAX], xq[MAX], yq[MAX];
  coef x[MAX], y[MAX];
  //  arrays
  coef* a_in;        //pointer to an array
  int  da_in;
  int dat, dbt, dq, dr;
  int dx1, dy1, dx2, dy2, dxq, dyq;                              //
degrees
  int dx, dy;
  int iloop;

  int poly_index;
  double sec1, sec2;
```

```
intro();

poly_read(num_polynomials, poly);

cout << "Your input file contains: " << endl;
poly_output(da, a);
cout << "---------the modulus." << endl;

const int MAX=80;
char filename[MAX];
ofstream fout;

cout << "\nEnter output file name: ";
cin.getline(filename, MAX, '\n');

fout.open(filename);

fout << "Results: " << endl;

poly_index = -1;

for(int kt=0; kt<nloop; kt++)
{

sec1 = clock();

for(int i=1; i<=kloop; i++) {

poly_index++;      //increase the index, should be zero, initially
if (poly_index >= num_polynomials)
  poly_index -= num_polynomials;          //ensure that it is less than the number used

a_in = &(poly[poly_index].bit[0]);                    //set a to point to the ith polynomial
da_in = poly[poly_index].deg;                                            //its degree

dx1=0;  x1[0]=ch1;
dy1=0; y1[0]=ch0;
dx2=0; x2[0]=ch0;
dy2=0; y2[0]=ch1;

poly_copy(da, a, dat, at);              // copy: a[] to at[], da to dat
poly_copy(da_in, a_in, dbt, bt);                // copy: b[] to bt[], db to dbt
iloop=0;

  while(1)
  {
```

```
        iloop++;
        poly_factor(dat, at, dbt, bt, dq, q, dr, r);            //  get q and r
        poly_multiply(dx2, x2, dq, q, dxq, xq);
        poly_multiply(dy2, y2, dq, q, dyq, yq);
        poly_add(dx1, x1, dxq, xq, dx, x);
        poly_add(dy1, y1, dyq, yq, dy, y);

        if(dr==0 && r[0]==ch1) break;
        if(dr==0 && r[0]==ch0)
        {
          cout << "The modulus is not irreducible" << endl;
          if(iloop==1) {
            cout << "The modulus is divisible by the inverse. The quotient is: ";
            poly_output(dq, q);
          }
          cout << "Please try again.\n" << endl;
          exit(1);
        }

        poly_copy(dx2, x2, dx1, x1);
        poly_copy(dy2, y2, dy1, y1);
        poly_copy(dx, x, dx2, x2);
        poly_copy(dy, y, dy2, y2);

        poly_copy(dbt, bt, dat, at);
        poly_copy(dr, r, dbt, bt);
      }
    }

    sec2 = clock();
    cout << "CPU time : " << 1.0*(sec2 - sec1) << " (ms) for " << kloop << " loops." <<
endl;
    fout << 1.0*(sec2 - sec1) << endl;
    }
    fout.close();

    poly_write(da, a, da_in, a_in, dx, x, dy, y);
    cout << "\n--------------------------------" << endl;
    cout << "Results: " << endl;
    cout << "The modulus is ";
    poly_output(da, a);
    cout << endl;
    cout << "The element is ";
    poly_output(da_in, a_in);
    cout << endl;
    cout << "\nThe multiplier of the modulus is: ";
    poly_output(dx, x);
```

```
  cout << endl;
  cout << "\nThe inverse is: ";
  poly_output(dy, y);
  cout << endl;

  return 0;
}


//------------------------------------------------------------------------
void poly_read(int num_polys, polytype* poly)
//  Read in degrees and coefficients of polynomials from a file  //
{
  int k;
  int i;;
  ifstream fin;
  ofstream fout;


  fin.open(infilename);

  for (k=0;k<num_polys;k++)
  {
    fin >> poly[k].deg;                        //  read degree of poly[]
    for (i=poly[k].deg; i>=0; i--)
    {
      fin >> poly[k].bit[i];            //  read coefficients of poly[]
//      poly[k].bit[i] &= 0X01;         //  only take low bit
    }
  }
}

//------------------------------------------------------------------------
void poly_write(int da, coef* a, int db, coef* b, int dx, coef *x, int dy, coef *y)
{
  const int MAXCHAR=80;
  char filename[MAXCHAR];
  ofstream fout;

  cout << "\nEnter output file name: ";
  cin.getline(filename, MAXCHAR, '\n');

  fout.open(filename);


  fout << "Results: " << endl;
  fout << "The modulus is ";
```

```
  poly_ofile(da, a, fout);
  fout << endl;
  fout << "The element is ";
  poly_ofile(db, b, fout);
  fout << endl;
  fout << "\nThe multiplier of the modulus is: ";
  poly_ofile(dx, x, fout);
  fout << endl;
  fout << "\nThe inverse is: ";
  poly_ofile(dy, y, fout);
  fout.close();
}

//--------------------------------------------------------------------------
void poly_factor(int da, coef *a, int db, coef *b, int &dq, coef *q, int &dr, coef *r)
{
  int d, i;

  dr=da, dq=da-db;                          //  degree = array index

  poly_copy(da, a, dr, r);
  poly_init(q);                  //  initialize q[] as zeros

  while(1) {
    d=dr-db;              //  d is a temporary exponent of a term of q
    q[d]=ch1;
    for(i=dr; i>=dr-db; i--) {
      if(r[i]==b[i-d]) r[i]=ch0;
      else r[i]=ch1;
    }

    dr=poly_degree(dr, r);

    if(dr<db) break;
  }

  return;
}

//--------------------------------------------------------------------------
void poly_multiply(int da, coef *a, int db, coef *b, int &dc, coef *c)
{
  int i, j, k;

  dc=da+db;
  poly_init(c);    //  initialization of c[]
  if((da==0 && a[0]==ch0) || (db==0 && b[0]==ch0)) {
```

```
    dc=0;
    return;
  }

  for(i=da; i>=0; i--) {
    for(j=db; j>=0; j--) {
      k=i+j;
      if(a[i]==ch1 && b[j]==ch1) {
        if(c[k]==ch0) c[k]=ch1;
        else if(c[k]==ch1) c[k]=ch0;
        else {
          cout << "Invalid coefficients in poly_multiply; program aborted.";
          exit(99);
        }
      }
    }
  }
}

//--------------------------------------------------------------------------
void poly_add(int da, coef *a, int db, coef *b, int &dc, coef *c)
{
  int i;

  poly_init(c);
  dc=da;
  if(db>da) dc=db;
  for(i=dc; i>=0; i--) {
    if(i<=da && i<=db) {
      if(a[i]==b[i]) c[i]=ch0;
      else c[i]=ch1;
    }
    else if(i<=da) c[i] = a[i];
    else if(i<=db) c[i] = b[i];
  }

  dc = poly_degree(dc, c);
}


//--------------------------------------------------------------------------
void poly_copy(int ds, coef *s, int &dt, coef *t)
{
  for(int i=0; i<MAX; i++)
    if(i<=ds) t[i]=s[i];
    else t[i]=ch0;
```

```
  dt=ds;
}

//--------------------------------------------------------------------------
void polytype_copy(polytype poly, int &dt, coef *t)
{
  for(int i=0; i<MAX; i++)
    if(i<=poly.deg) t[i]=poly.bit[i];
    else t[i]=ch0;

  dt=poly.deg;
}

//--------------------------------------------------------------------------
int poly_degree(int d, coef *p)
{
  int i, j;

  for(i=d; i>=0; i--) {                    //  find out the degree of r[], which is dr
    j=i;
    if(p[i]!=ch0) break;
  }
  return j;
}

//--------------------------------------------------------------------------
void poly_output(int d, coef *p)
{
  if(d==0)
    cout << p[d] << endl;
  else {
    if(d==1 && p[0]==ch1) cout << "\nx + 1" << endl;
    else if(d==1 && p[0]==ch0) cout << "\nx" << endl;
    else {
      cout << "\nx^" << d;
      for(int i=d-1; i>=2; i--) {
        if(p[i]==ch1) cout << " + " << "x^" << i;
      }
      if(p[1]==ch1) cout << " + x";
      if(p[0]==ch1) cout << " + 1";
      cout << endl;
    }
  }
}

//--------------------------------------------------------------------------
void poly_ofile(int d, coef *inverse, ofstream &fout)
```

```
{
  int k=0;                    //  k is a counter

  if(d==0)
    fout << inverse[d] << endl;
  else {
    if(d==1 && inverse[0]==ch1) fout << "\nx + 1" << endl;
    else if(d==1 && inverse[0]==ch0) fout << "\nx" << endl;
    else {
      fout << "\n   x^" << d; k++;
      for(int i=d-1; i>=2; i--) {
        if(inverse[i]==ch1) {

          if(i>=100)    fout << " + " << "x^" << i;
          else if(i>=10)          fout << " +  " << "x^" << i;
          else        fout << " +   " << "x^" << i;

          k++;
          if(k%10==0) fout << endl;
        }
      }

      if(inverse[1]==ch1) {
        fout << " +     x";
        k++;
        if(k%10==0) fout << endl;
      }
      if(inverse[0]==ch1) fout << " +     1";
      fout << endl;
    }
  }
}

//--------------------------------------------------------------------------
void poly_init(coef *s)
{
  for(int i=0; i<MAX; i++) s[i]=ch0;
  return;
}

//--------------------------------------------------------------------------
void intro()
{
  cout << "Generate an input-file that contains the degrees and the coefficients " << endl;
  cout << "of two binary polynomials. First the modulus, followed by the inverse." <<
endl;
  cout << "\nFor example: " << endl;
```

```
  cout << "If the modulus is x^3 + x + 1, and the inverse is x^2 + 1," << endl;
  cout << "then the input file should be the following: " << endl;
  cout << "\n3\n1 0 1 1\n2\n1 0 1" << endl;
  cout << "\nThe results will be both printed to the screen and saved to a user-given file."
<< endl;

  return;
}
```

Extended Euclidean Algorithm, Byte Arrays

```
/*
  EEA_bytes.cpp
  Extended Euclidean Algorithm, but not with multiplicand table
  8 coefficients per array element
  Chen Zhao & Kristin Cordwell
  03 April 2006
*/

#include <iostream>
#include <fstream>
#include <ctime>

using namespace std;

const int NBYTES=20;                        //  number of bytes for each
polynomial
const int MAXCHAR=80;                       //  maximum length of filenames

typedef struct
{
  unsigned char coeff[NBYTES];
  int  deg;
} poly_type;

struct two_byte_type
{
  unsigned char byte[2];
};

two_byte_type table[256][256];             //  table for multiplying result

void poly_read(poly_type*, ifstream&);
void shift(poly_type*, int, poly_type*);
int poly_degree(poly_type *);
void poly_copy(poly_type*, poly_type*);
void mult_table(two_byte_type t[][256]);
void poly_multiply(poly_type*, poly_type*, poly_type*);
void poly_output(poly_type*, ofstream&);
void output_byte(unsigned char);

//============================================================
==============
int main()
{
```

```
ifstream fin;
ofstream fout;
char inpfname [] = "poly_list_EEA.txt";     //  fixed input list of random polynomials
char outpfname [] = "poly_output.txt";      //  fixed output list of results
char cpufname[] = "cpu_times.txt";
int seconds1, seconds2;
int Ntrials;

int i, j, dd, c;
int kpoly;

poly_type *a;                               //  used to access list of polynomials
poly_type b, quot[155], rem[3], qx2, qy2;

poly_type x10 = { 0X01, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00,
            0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0};
poly_type y10 = { 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00,
            0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0};
poly_type x20 = { 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00,
            0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0};
poly_type y20 = { 0X01, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00,
            0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0};

poly_type p = { 0X01, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X40, 0X00, 0X00,
          0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X08, 155};
poly_type x1, x2, x3, y1, y2, y3;
mult_table(table);

cout << "How many trials? ";
cin >> Ntrials;


fout.open(outpfname);
fin.open(inpfname);
fin >> kpoly;
a = new poly_type[kpoly];

cout << "Reading list of polynomials from file " << inpfname << " ......" << endl;
for(i=1; i<=kpoly; i++) {
  poly_read(&a[i-1], fin);
  fout << "Input polynomial #" << i << " is: " << endl;
  poly_output(&a[i-1], fout);
}
cout << "Finished reading" << endl;
fin.close();

for(int it=1; it<=Ntrials+1; it++) {
```

```
seconds1=clock();
for(int ip=1; ip<=kpoly; ip++) {
  x1 = x10;           //  re-initialize x1, x2, y1, y2
  y1 = y10;
  x2 = x20;
  y2 = y20;

  c=0;                //  c is the number of (quot)s
  rem[0] = p;
  rem[1] = a[ip-1];

  while(1) {
    for(i=0; i<NBYTES; i++) quot[c].coeff[i]=0X00;          //  initialize quot

    dd = rem[0].deg - rem[1].deg;
    quot[c].deg=dd;

    rem[2]=rem[0];

    while(dd>=0) {

      quot[c].coeff[dd/8] ^= (0X01<<(dd%8));                //  put x^dd to quot
      shift(&rem[1], dd, &b);
      for(i=0; i<=19; i++) rem[2].coeff[i]^=b.coeff[i];
      rem[2].deg = poly_degree(&rem[2]);
      dd = rem[2].deg - rem[1].deg;
    }

    rem[0]=rem[1];
    rem[1]=rem[2];
    c++;
    if(rem[2].deg==0) break;
  }

  for(i=0; i<c; i++) {
    poly_multiply(&quot[i], &x2, &qx2);
    poly_multiply(&quot[i], &y2, &qy2);
    for(j=0; j<NBYTES; j++) {
      x3.coeff[j] = qx2.coeff[j] ^ x1.coeff[j];
      y3.coeff[j] = qy2.coeff[j] ^ y1.coeff[j];
    }
    x3.deg = poly_degree(&x3);
    y3.deg = poly_degree(&y3);

    x1=x2; x2=x3;
    y1=y2; y2=y3;
  }
```

```
    if(it==Ntrials+1) {
      if(ip==1) cout << "Writing results to file......" << endl;
      fout << "Inverse polynomial #" << ip << " is:  " << endl;
      poly_output(&y3, fout);
    }
  }

  if(it<=Ntrials) {
    seconds2=clock();
    cout << "CPU Time = " << seconds2 - seconds1 << " (ms) " <<
      "for " << kpoly << " polynomials " << endl;
  }
  }

  return 0;
}

//-------------------------------------------------------------------
void shift(poly_type* a, int ishift, poly_type *b)
{
  int ibsh = ishift/8;
  int ibit = ishift%8;
  int j;

  for (j=0; j<NBYTES; j++) (*b).coeff[j] = 0X00;

  for (j=0; j<NBYTES; j++) {
    (*b).coeff[j+ibsh+0] ^= (*a).coeff[j] << ibit;
    (*b).coeff[j+ibsh+1] ^= (*a).coeff[j] >> (8-ibit);
  }
  (*b).deg = (*a).deg + ishift;
}

//-------------------------------------------------------------------
void poly_read(poly_type* a, ifstream &fin)
//  Read degree and coefficients (int 1 or 0) of a polynomial from an input file
//  Convert int to bit (1/8 of a byte) and store in an array of unsigned char
//  a[] are the coefficients of the polynomial, a_deg is the degree of a[]
{
  int i;
  int coeff;
  int hi_byte;          //  highest index of bytes
  int hi_bit;           //  highest index of bit in hi_byte


  for(i=0; i<NBYTES; i++)           // initialize all bytes
    (*a).coeff[i] = 0X00;
```

```
  fin >> (*a).deg;                    //  read degree of the polynomial
  hi_byte=(*a).deg/8;
  hi_bit=(*a).deg%8;

  for (i=hi_bit; i>=0; i--) {                 //  read the first partial byte
    fin >> coeff;
    if(coeff==1)
      (*a).coeff[hi_byte]=(*a).coeff[hi_byte]^(0X01<<i);
  }

  for (int ib=hi_byte-1; ib>=0; ib--) {            //  read the rest full bytes
    for (i=7; i>=0; i--) {                                          //  read 8 bits (a byte)
      fin >> coeff;
      if(coeff==1)
        (*a).coeff[ib] ^= (0X01<<i);
    }
  }
}

//---------------------------------------------------------------------
int poly_degree(poly_type *a)
{
  int ib;
  for(int j=19; j>=0; j--) {
    ib=j;
    if((*a).coeff[j]!=0X00) break;
  }
  for(int k=7; k>=0; k--) if((((*a).coeff[ib]>>k) & 0X01) == 0X01) break;
  return 8*ib+k;
}

//---------------------------------------------------------------------
void poly_copy(poly_type* sender, poly_type* receiver)
{
  for(int i=0; i<=19; i++) (*receiver).coeff[i]=(*sender).coeff[i];
  (*receiver).deg=(*sender).deg;
}

//---------------------------------------------------------------------
void mult_table(two_byte_type table[][256])
//construct table of multiplication elements, byte pairs
{
  int  i, j;
  unsigned char a, b;
  unsigned char product[2];
```

```
  for (i=0; i<256; i++) {
    a = (unsigned char) i;
    for (j=0; j<256; j++) {
      b = (unsigned char) j;
      product[0]=0X00; product[1]=0X00;                    //  initialization of product
      for(int k=0; k<8; k++) {
        if(((b>>k) & 0X01) == 0X01) {
          product[0] ^= a << k;
          product[1] ^= a >> (8-k);
        }
      }
      table[i][j].byte[0] = product[0] & 0XFF;                          //low byte
      table[i][j].byte[1] = product[1] & 0XFF;                          //high byte
    }
  }

  return;
}

//---------------------------------------------------------------------
void poly_multiply(poly_type* a, poly_type *b, poly_type* c)
{
  int ai;
  int bj;

  for(int i=0; i<NBYTES; i++)
    (*c).coeff[i]=0X00;        //  initialize polynomial c
  (*c).deg=(*a).deg+(*b).deg;
  //  calculuate degree of c

  for(i=0; i<=(*a).deg/8; i++) {
    ai = (int) (*a).coeff[i];
    for(int j=0; j<=(*b).deg/8; j++) {
      bj = (*b).coeff[j];
      (*c).coeff[i+j] ^= table[ai][bj].byte[0];
      (*c).coeff[i+j+1] ^= table[ai][bj].byte[1];
    }
  }
}

//---------------------------------------------------------------------
void poly_output(poly_type* poly, ofstream &fout)                  //  output one
polynomial (poly) to screen and to file
{
  int hi_byte=(*poly).deg/8;
  int hi_bit=(*poly).deg%8;
  int count=1;
```

```
  int d;

  if((*poly).deg==0 && (*poly).coeff[0]==0X01) fout << "   1";              //  for poly of
degree =0
  else if((*poly).deg==1) {
    if(((*poly).coeff[0]&0X01)==0X01) fout << " x   + 1";                    //  for poly of
degree =1
    else fout << " x";
  } else {                                                                    //  for
poly of degree >=2
    fout << "   x^" << (*poly).deg;
    for(int i=hi_bit-1; i>=0; i--) {                                   // the hi_byte (first
partial byte)
      d=8*hi_byte+i;

      if((((*poly).coeff[hi_byte]>>i)&0X01)==0X01) {
        if(d==0)    fout << "   + 1";
        else if(d==1)fout << " + x";
        else {
          fout << " + x^" << d;                    //  output to file
          count++;
        }
      }
    }

    for(int j=hi_byte-1; j>=0; j--) {                                 // the rest full byte
      for(int k=7; k>=0; k--) {
      if((((*poly).coeff[j]>>k)&0X01)==0X01) {
        d=8*j+k;
        if(d==0)       fout << "   + 1";
        else if(d==1)   fout << " + x   ";
        else {
          if(d>=100)    fout << " + x^" << d;
          else if (d>=10) fout << " + x^" << d << " ";
          else         fout << " + x^" << d << "   ";
        }
        count++;
        if(count%10==0) fout << endl;
      }
      }
    }
  }
  fout << endl << endl;
}

//-----------------------------------------------------------------
void output_byte(unsigned char b)
```

```
{
  int bit;
  for (int i=7; i>=0; i--) {
    bit = (b>>i) & 0X01;
    cout << bit;
  }
  cout << endl << endl;
}
```

Extended Euclidean Algorithm, Byte Arrays, Multiplication and Reducation Tables

```
/*
  EEA_tables_bytes.cpp
  Extended Euclidean Algorithm, with table for multiplicand
  8 coefficients per array element
  Chen Zhao & Kristin Cordwell
  4-3-06
*/

#include <iostream>
#include <fstream>
#include <ctime>

using namespace std;

const int NBYTES=20;                    //  number of bytes for each
polynomial
const int MAXCHAR=80;                   //  maximum length of filenames

typedef struct
{
  unsigned char coeff[NBYTES];
  int  deg;
} poly_type;

struct two_byte_type
{
  unsigned char byte[2];
};

two_byte_type table[256][256];          //  table for multiplying result

void poly_read(poly_type*, ifstream&);
void shift(poly_type*, int, poly_type*);
int poly_degree(poly_type *);
void poly_copy(poly_type*, poly_type*);
void mult_table(two_byte_type t[][256]);
void poly_multiply(poly_type*, poly_type*, poly_type*);
void poly_output(poly_type*, ofstream&);
void output_byte(unsigned char);
int  high_bit(unsigned char);
unsigned char  multiplicand(unsigned char , unsigned char);
void  make_table1(void);

unsigned char  m_table[256][256];
//lookup table for multiplicands
```

```
//==================================================================
======================
int main()
{

   ifstream fin;
   ofstream fout;
   char inpfname [] = "poly_list_EEA.txt";     //  fixed input list of random polynomials
   char outpfname [] = "poly_output.txt";      //  fixed output list of results
   char cpufname[] = "cpu_times.txt";
   int seconds1, seconds2;
   int Ntrials;
   unsigned char  multiplicand;

   int i, j, dd, c;
   int kpoly;

   int  dd_over_8;

   poly_type *a;                               //  used to access list of polynomials
   poly_type quot[155], rem[3], qx2, qy2;

   poly_type x10 = { 0X01, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00,
              0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0};
   poly_type y10 = { 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00,
              0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0};
   poly_type x20 = { 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00,
              0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0};
   poly_type y20 = { 0X01, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00,
              0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0};

   poly_type p = { 0X01, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X40, 0X00, 0X00,
           0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X08, 155};
   poly_type x1, x2, x3, y1, y2, y3;

   make_table1();        //make quotient (byte) table
   mult_table(table); //make multiplication table

   cout << "How many trials? ";
   cin >> Ntrials;


   fout.open(outpfname);
   fin.open(inpfname);
   fin >> kpoly;
   a = new poly_type[kpoly];
```

```
  cout << "Reading list of polynomials from file " << inpfname << " ......" << endl;
  for(i=1; i<=kpoly; i++) {
    poly_read(&a[i-1], fin);
    fout << "Input polynomial #" << i << " is: " << endl;
    poly_output(&a[i-1], fout);
  }
  cout << "Finished reading " << kpoly << " polynomials" << endl;
  fin.close();

  for(int it=1; it<=Ntrials+1; it++)
  {
  seconds1=clock();

  for(int ip=1; ip<=kpoly; ip++)
  {
    x1 = x10;           //  re-initialize x1, x2, y1, y2
    y1 = y10;
    x2 = x20;
    y2 = y20;

    c=0;            //  c is the number of (quot)s
    rem[0] = p;
    rem[1] = a[ip-1];

    while(1)
    {
      for(i=0; i<NBYTES; i++)
        quot[c].coeff[i]=0X00;              //  initialize quot

      dd = rem[0].deg - rem[1].deg;
      quot[c].deg=dd;

      rem[2]=rem[0];

      while (dd >= 0)
      {
        multiplicand =
          m_table[int(rem[2].coeff[(rem[2].deg)/8])][int(rem[1].coeff[(rem[1].deg)/8])];
//look up multiplicand for high bytes
        dd_over_8 = dd/8;
        quot[c].coeff[dd_over_8] ^= multiplicand;

        for (i=0; i<=(rem[1].deg/8); i++)
        {
          rem[2].coeff[i + dd_over_8] ^= table[rem[1].coeff[i]][multiplicand].byte[0];
          rem[2].coeff[i + 1 + dd_over_8] ^= table[rem[1].coeff[i]][multiplicand].byte[1];
```

```
         }
//note--this might overrun the array bounds, but will be XORing 0 in that case


         rem[2].deg = poly_degree(&rem[2]);
         dd = rem[2].deg - rem[1].deg;
       }

       rem[0]=rem[1];
       rem[1]=rem[2];
       c++;
       if(rem[2].deg==0) break;
     } //end while (1)

     for(i=0; i<c; i++)
     {
       poly_multiply(&quot[i], &x2, &qx2);
       poly_multiply(&quot[i], &y2, &qy2);
       for(j=0; j<NBYTES; j++)
       {
         x3.coeff[j] = qx2.coeff[j] ^ x1.coeff[j];
         y3.coeff[j] = qy2.coeff[j] ^ y1.coeff[j];
       }
       x3.deg = poly_degree(&x3);
       y3.deg = poly_degree(&y3);

       x1=x2; x2=x3;
       y1=y2; y2=y3;
     }
     if(it==Ntrials+1)
     {
       if(ip==1) cout << "Writing results to file......" << endl;
       fout << "Inverse polynomial #" << ip << " is:  " << endl;
       poly_output(&y3, fout);
     }
   }

   if(it<=Ntrials)
   {
     seconds2=clock();
     cout << "CPU Time = " << seconds2 - seconds1 << " (ms) "
       << "for " << kpoly << " polynomials" << endl;
   }

   }

   return 0;
```

```
}

//------------------------------------------------------------------
int  high_bit(unsigned char x)
{
  int i;

  i = 0;

  while (x!=0)
  {
    x = x >> 1;
    i++;
  }

  return (i-1);  //count from 0, -1 if input is identically 0
}

//------------------------------------------------------------------
unsigned char  multiplicand(unsigned char a, unsigned char b)
{
  int  a_high, b_high;
  unsigned char temp;
  int  delta;

  temp = 0;

  a_high = high_bit(a);
  b_high = high_bit(b);

  delta = a_high - b_high;

  if (int(a_high) >= int(b_high))
    while (int(a_high) >= int(b_high))
    {
      a = a ^ (b << delta);
      temp = temp | (0X01 << delta);
      a_high = high_bit(a);
      delta = a_high - b_high;
    }
  else
    while ((int(b_high) > int(a_high)) && (a != 0))
    {
      a = a ^ (b >> (-delta));
      temp = temp | (0X01 << (8 + delta));
      a_high = high_bit(a);
      delta = a_high - b_high;
```

```
    }

  return temp;
}

//---------------------------------------------------------------------
void  make_table1(void)
//takes the high bytes of polynomials and calculates a multiplicand
{
  int i,j;

  for (i=0; i<256; i++)
    for (j=0; j< 256; j++)
      m_table[i][j] = 0X00;

  for (i=1; i<256; i++)
    for (j=1; j< 256; j++)
      m_table[i][j] = multiplicand((unsigned char) i, (unsigned char) j);

  return;
}

//----------------------------------------------------------------------
void shift(poly_type* a, int ishift, poly_type *b)
{
  int ibsh = ishift/8;
  int ibit = ishift%8;
  int j;

  for (j=0; j<NBYTES; j++) (*b).coeff[j] = 0X00;

  for (j=0; j<NBYTES; j++) {
    (*b).coeff[j+ibsh+0] ^= (*a).coeff[j] << ibit;
    (*b).coeff[j+ibsh+1] ^= (*a).coeff[j] >> (8-ibit);
  }
  (*b).deg = (*a).deg + ishift;
}


//---------------------------------------------------------------------
void poly_read(poly_type* a, ifstream &fin)
//  Read degree and coefficients (int 1 or 0) of a polynomial from an input file
//  Convert int to bit (1/8 of a byte) and store in an array of unsigned char
//  a[] are the coefficients of the polynomial, a_deg is the degree of a[]
{
  int i;
  int coeff;
```

```
  int hi_byte;             //  highest index of bytes
  int hi_bit;              //  highest index of bit in hi_byte


  for(i=0; i<NBYTES; i++)              // initialize all bytes
    (*a).coeff[i] = 0X00;

  fin >> (*a).deg;                 //  read degree of the polynomial
  hi_byte=(*a).deg/8;
  hi_bit=(*a).deg%8;

  for (i=hi_bit; i>=0; i--) {               //  read the first partial byte
    fin >> coeff;
    if(coeff==1)
      (*a).coeff[hi_byte]=(*a).coeff[hi_byte]^(0X01<<i);
  }

  for (int ib=hi_byte-1; ib>=0; ib--) {           //  read the rest full bytes
    for (i=7; i>=0; i--) {                                          //  read 8 bits (a byte)
      fin >> coeff;
      if(coeff==1)
        (*a).coeff[ib] ^= (0X01<<i);
    }
  }
}


//-----------------------------------------------------------------------
int poly_degree(poly_type *a)
{
  int ib;
  for(int j=19; j>=0; j--) {
    ib=j;
    if((*a).coeff[j]!=0X00) break;
  }
  for(int k=7; k>=0; k--) if((((*a).coeff[ib]>>k) & 0X01) == 0X01) break;
  return 8*ib+k;
}


//-----------------------------------------------------------------------
void poly_copy(poly_type* sender, poly_type* receiver)
{
  for(int i=0; i<=19; i++) (*receiver).coeff[i]=(*sender).coeff[i];
  (*receiver).deg=(*sender).deg;
}
```

```
//--------------------------------------------------------------------
void mult_table(two_byte_type table[][256])
//construct table of multiplication elements, byte pairs
{
   int  i, j;
   unsigned char a, b;
   unsigned char product[2];

   for (i=0; i<256; i++) {
      a = (unsigned char) i;
      for (j=0; j<256; j++) {
         b = (unsigned char) j;
         product[0]=0X00; product[1]=0X00;                    //  initialization of product
         for(int k=0; k<8; k++) {
            if(((b>>k) & 0X01) == 0X01) {
               product[0] ^= a << k;
               product[1] ^= a >> (8-k);
            }
         }
         table[i][j].byte[0] = product[0] & 0XFF;                       //low byte
         table[i][j].byte[1] = product[1] & 0XFF;                       //high byte
      }
   }

   return;
}


//--------------------------------------------------------------------
void poly_multiply(poly_type* a, poly_type *b, poly_type* c)
{
   int ai;
   int bj;

   for(int i=0; i<NBYTES; i++)
      (*c).coeff[i]=0X00;          //  initialize polynomial c
   (*c).deg=(*a).deg+(*b).deg;
   //  calculuate degree of c

   for(i=0; i<=(*a).deg/8; i++) {
      ai = (int) (*a).coeff[i];
      for(int j=0; j<=(*b).deg/8; j++) {
         bj = (*b).coeff[j];
         (*c).coeff[i+j] ^= table[ai][bj].byte[0];
         (*c).coeff[i+j+1] ^= table[ai][bj].byte[1];
      }
```

```
  }
}


//-------------------------------------------------------------------
void poly_output(poly_type* poly, ofstream &fout)
//  output one polynomial (poly) to screen and to file
{
  int hi_byte=(*poly).deg/8;
  int hi_bit=(*poly).deg%8;
  int count=1;
  int d;

  if((*poly).deg==0 && (*poly).coeff[0]==0X01) fout << "   1";              //  for poly of
degree =0
  else if((*poly).deg==1) {
    if(((*poly).coeff[0]&0X01)==0X01) fout << " x   + 1";                     //  for poly of
degree =1
    else fout << " x";
  } else {                                                                    //  for
poly of degree >=2
    fout << "   x^" << (*poly).deg;
    for(int i=hi_bit-1; i>=0; i--) {                                  // the hi_byte (first
partial byte)
      d=8*hi_byte+i;

      if((((*poly).coeff[hi_byte]>>i)&0X01)==0X01) {
        if(d==0)    fout << "   + 1";
        else if(d==1)fout << " + x";
        else {
          fout << " + x^" << d;                    //  output to file
          count++;
        }
      }
    }

    for(int j=hi_byte-1; j>=0; j--) {                                 // the rest full byte
     for(int k=7; k>=0; k--) {
      if((((*poly).coeff[j]>>k)&0X01)==0X01) {
        d=8*j+k;
        if(d==0)       fout << "   + 1";
        else if(d==1)   fout << " + x   ";
        else {
          if(d>=100)   fout << " + x^" << d;
          else if (d>=10) fout << " + x^" << d << " ";
          else        fout << " + x^" << d << "  ";
        }
```

```
            count++;
            if(count%10==0) fout << endl;
          }
        }
      }
  }
  fout << endl << endl;
}


//----------------------------------------------------------------------
void output_byte(unsigned char b)
{
  int bit;
  for (int i=7; i>=0; i--) {
    bit = (b>>i) & 0X01;
    cout << bit;
  }
  cout << endl << endl;
}
```

Lagrange's Theorem (Algorithm), Russian Peasant Multiplication

```
/*
   Lagrange's Theorem for Polynomials
     Russian Peasant Method
     1 bit per array element
   Chen Zhao, Kristin Cordwell
   02-11-06
*/

#include <iostream>
//#include <fstream>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <fstream>

using namespace std;

// modulus binary trinomial is x^155 + x^62 + 1
char ch0='0';
char ch1='1';

const unsigned int PMAX=310;                    // PMAX-1 = max degree of
polynomial
const unsigned int DMOD=155;                    // degree of modulus
const unsigned int R1=DMOD-0;                   // used for reduction
const unsigned int R2=DMOD-62;                  // ditto

void input(int&, int*);
void poly_read(int&, int*);
void convert(int, int*, char*);
void square(int&, char*);
void poly_multiply(int&, char*, int, char*);
void reduce(int&, char*);
int poly_degree(int, char*);
void output(int, char*);
void poly_write(int, char*, int, char*);
void poly_ofile(int, char*, ofstream&);
void init(int, char*);          //  Character only
void poly_copy(int, char*, int&, char*);

//============================================================
============
int main()
{
  int d, dbk;
```

```
  int p[PMAX];

   char inverse[PMAX];
  char backup[PMAX];

  int nloop=100;
  long seconds1;
  long seconds2;

  for(int i=0; i<PMAX; i++) p[i]=0;                    //  Initialization of p[], integer
  for(int j=0; j<PMAX; j++) inverse[j]=ch0; //  Initialization of inverse[], char

  //input(d, p);                               // user input inverse as an array of integers
  poly_read(d, p);                                    // user input inverse from a file

  dbk=d;                              //  Original degree
  convert(d, p, backup);                                          //  convert from
integer to char
   convert(d, p, inverse);                                        // ditto

  seconds1 = clock();                               // return number of clock ticks at this
time point

  for(int nt=1; nt<=nloop; nt++)
  {
    poly_copy(dbk, backup, d, inverse);
    for(i=1; i<=DMOD-1; i++) {
    //for(i=1; i<=10; i++) {                               // specific number of
iterations
       square(d, inverse);
       while(d>=DMOD) reduce(d, inverse);
       if(i<=DMOD-2) poly_multiply(d, inverse, dbk, backup);
       while(d>=DMOD) reduce(d, inverse);
    }
  }

  seconds2 = clock();                          // return number of clock ticks at this time
point
  //cout << "CLOCKS_PER_SEC = " << CLOCKS_PER_SEC << endl;
  cout << "CPU time : " << 1.0*(seconds2 - seconds1)/CLOCKS_PER_SEC << "(s) for "
<< nloop << " loops." << endl;

  //output(d, inverse);                                  // output to screen
  poly_write(dbk, backup, d, inverse);                                       //
output to file

  return 0;
```

```
}

//-------------------------------------------------------------------
void poly_read(int &d, int *p)
{
  int i;
  const int MAXCHAR=80;
  char filename[MAXCHAR];
  ifstream fin;
  ofstream fout;

  cout << "\nEnter input file name: ";
  cin.getline(filename, MAXCHAR, '\n');

  fin.open(filename);

  fin >> d;          //  read degree of p[]
  for (i=d; i>=0; i--)
    fin >> p[i];   //  read coefficients of p[]
}

//-------------------------------------------------------------------
void input(int &d, int *p)
{
  cout << "Input the degree of the inverse polynomial:  ";
  cin >> d;

   cout << "\nInput coefficients (must be 0 or 1): " << endl;
  for(int i=d; i>0; i--) {
    cout << "\nx^" << i << ": ";
    cin >> p[i];
  }
  cout << "\nconstant: ";
  cin >> p[0];

   for(int j=0; j<=d; j++) {
      if(p[j]!=0 && p[j]!=1) {
       cout << "Bad number detected" << endl;
       cout << "Program aborted..." << endl;
       exit(264);
     }
   }
}

//-------------------------------------------------------------------
void convert(int d, int *p, char *inverse)
{
```

```
    for(int i=0; i<=d; i++) {
        if(p[i]==0) inverse[i]=ch0;
            else inverse[i]=ch1;
    }
}

//----------------------------------------------------------------------
void square(int &d, char *inverse)                                    // return d and inverse
{
   char tempp[PMAX];
   init(PMAX-1, tempp);                                                              //
Initialization of tempp[]
    for(int i=0; i<=d; i++) {
        if(inverse[i]==ch1) tempp[2*i]=ch1;
   }
   d*=2;                                                       // new degree
   for(int j=0; j<=d; j++) inverse[j]=tempp[j];           //  Transfer tempp[] back to inverse[]
}

//----------------------------------------------------------------------
void reduce(int &d, char *inverse)
{
   for(int i=DMOD; i<=d; i++) {
     if(inverse[i]==ch1) {
       inverse[i]=ch0;
       if(inverse[i-R1]==ch0) inverse[i-R1]=ch1;
       else inverse[i-R1]=ch0;
       if(inverse[i-R2]==ch0) inverse[i-R2]=ch1;
       else inverse[i-R2]=ch0;
     }
   }
   d=poly_degree(d, inverse);
}

//----------------------------------------------------------------------
void poly_multiply(int& da, char *a, int db, char *b)
{
   int i, j, k;
   int dc;
   char c[PMAX];

   dc=da+db;
   init(dc, c);                                                       //  initialization of c[]
   if((da==0 && a[0]==ch0) || (db==0 && b[0]==ch0)) return;        //  If both polynomials
are 0, the product will also be 0.

   for(i=da; i>=0; i--) {
```

```
    for(j=db; j>=0; j--) {
      k=i+j;
      if(a[i]==ch1 && b[j]==ch1) {
        if(c[k]==ch0) c[k]=ch1;
        else if(c[k]==ch1) c[k]=ch0;
        else {
          cout << "Invalid coefficients in poly_multiply; program aborted." << endl;
          exit(99);
        }
      }
    }
  }

  for(i=dc; i>=0; i--) a[i]=c[i];
  da=dc;
}

//-----------------------------------------------------------------------
int poly_degree(int d, char *inverse)
{
  int i, j;

  for(i=d; i>=0; i--) {                    //  find out the degree of inverse[], which is j
    j=i;
    if(inverse[i]!=ch0) break;
  }
  return j;
}

//-----------------------------------------------------------------------
void output(int d, char *inverse)
{
  if(d==0)
    cout << inverse[d] << endl;
  else {
    if(d==1 && inverse[0]==ch1) cout << "\nx + 1" << endl;
    else if(d==1 && inverse[0]==ch0) cout << "\nx" << endl;
    else {
      cout << "\nx^" << d;
      for(int i=d-1; i>=2; i--) {
        if(inverse[i]==ch1) cout << " + " << "x^" << i;
      }
      if(inverse[1]==ch1) cout << " + x";
      if(inverse[0]==ch1) cout << " + 1";
      cout << endl;
    }
  }
```

```
  cout << endl;
}

//-----------------------------------------------------------------
void poly_write(int dbk, char* backup, int d, char *inverse)
{
  const int MAXCHAR=80;
  char filename[MAXCHAR];
  ofstream fout;

  cout << "\nEnter output file name: ";
  cin.getline(filename, MAXCHAR, '\n');

  fout.open(filename);

  fout << "The element is ";
  poly_ofile(dbk, backup, fout);
  fout << endl << endl;

  fout << "Results: " << endl;
  fout << "\nThe inverse is ";
  poly_ofile(d, inverse, fout);
  fout.close();
}

//-----------------------------------------------------------------
void poly_ofile(int d, char *inverse, ofstream &fout)
{
  int k=0;                //  k is a counter

  if(d==0)
    fout << inverse[d] << endl;
  else {
    if(d==1 && inverse[0]==ch1) fout << "\nx + 1" << endl;
    else if(d==1 && inverse[0]==ch0) fout << "\nx" << endl;
    else {
      fout << "\n   x^" << d; k++;
      for(int i=d-1; i>=2; i--) {
        if(inverse[i]==ch1) {

          if(i>=100)    fout << " + " << "x^" << i;
          else if(i>=10)         fout << " +  " << "x^" << i;
          else        fout << " +   " << "x^" << i;

          k++;
          if(k%10==0) fout << endl;
        }
```

```
        }

        if(inverse[1]==ch1) {
          fout << " +  x";
          k++;
          if(k%10==0) fout << endl;
        }
        if(inverse[0]==ch1) fout << " +  1";
        fout << endl;
      }
    }
}

//--------------------------------------------------------------------
void init(int d, char *p)
{
  for(int i=0; i<=d; i++) p[i]=ch0;
}

//--------------------------------------------------------------------//----------------------------------
------------------------------------
void poly_copy(int ds, char *s, int &dt, char *t)
{
  for(int i=0; i<PMAX; i++)
    if(i<=ds) t[i]=s[i];                    // copy array s[] to t[]
    else t[i]=ch0;

  dt=ds;
}
```

Lagrange’s Theorem (Algorithm), Itoh’s Technique, Bit Arrays

```
/*
   Lagrange's Theorem for Polynomials
     Single Coefficient per Array Element
     Itoh's Method
   Chen Zhao, Kristin Cordwell
   03-16-06
*/

#include <iostream>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <fstream>

using namespace std;

// modulus binary trinomial is x^155 + x^62 + 1
char ch0='0';
char ch1='1';

const unsigned int PMAX=310;                    // PMAX-1 = max degree of
polynomial
const unsigned int DMOD=155;                    // degree of modulus
const unsigned int R1=DMOD-0;                   // used for reduction (R1 =
155)
const unsigned int R2=DMOD-62;                  // ditto              (R2 =
93)

const int MAXCHAR=80;                                  // maximum length of
filenames

static double T_MULT=0.0;

const int MAX_NUM_POLYS = 50;
int Ntrials=50;
int npoly=50;
int kpoly=1*npoly;
//  read and use this number of polynomials from the input file, up to
MAX_NUM_POLYS

typedef struct
{
  char  coeff[PMAX];
  int    deg;
} poly_type;
```

```
poly_type  poly[MAX_NUM_POLYS];                    //list of random polynomials

void input(int&, int*);
void poly_read(int&, int*, ifstream&);
void convert(int, int*, char*);
void square(int&, char*);
void square_ntimes(int, int&, char*);
void poly_multiply(int&, char*, int, char*);
void reduce(int&, char*);
void reduce2(int&, char*);
int poly_degree(int, char*);
void output(int, char*);
void poly_ofile(int, char*, int, char*, ofstream&);
void init(int, char*);              //  Character only
void poly_copy(int, char*, int&, char*);

char*a;            //used to access list of polynomials
int  a_int[PMAX]; //used to access list of poynomials
int  a_deg;          //for degree of a

char  inpfname [] = "poly_list.txt";              //fixed input list of random polynomials

int main()
{
   int dq, dbk, dtp;
   int qint[PMAX];
   int  poly_index;       //which random polynomial to point to

   char q[PMAX];
   char temp[PMAX];

   long seconds1;
   long seconds2;

   ifstream fin;
   ofstream fout_cpu;
   ofstream fout_ans;

   char filename[MAXCHAR];
// char inpfname[MAXCHAR];

   for(int i=0; i<PMAX; i++) qint[i]=0;                   //  Initialization of qint[], integer
   for(int j=0; j<PMAX; j++) q[j]=ch0;                              //  Initialization of inverse[],
char

// cout << "\nEnter input file name: ";           //  the file that contains polynomials
```

```
// cin.getline(inpfname, MAXCHAR, '\n');


  cout << "\nEnter output file name for CPU Times: ";                 // for CPU recording
  cin.getline(filename, MAXCHAR, '\n');
  fout_cpu.open(filename);
  fout_cpu << "CPU time: " << endl;

  cout << "\nEnter output file name for answer: ";              // for answers (inverse)
  cin.getline(filename, MAXCHAR, '\n');
  fout_ans.open(filename);
  fout_ans << "Results: " << endl;

//first, read in the random polynomials
    fin.open(inpfname);                              //random polynomial input file
    for (i=0; i<npoly; i++)
    {
      poly_read(a_deg, a_int, fin);                          //  read input poly from a file
      dbk = a_deg;
  //  Original degree
      a = &(poly[i].coeff[0]);                                    //  set a to point to the ith
polynomial storage location
      convert(a_deg, a_int, a);                                           //  convert from
integer to char coefficient storage
      poly[i].deg = a_deg;                                                    //  set the
degree
    }
    fin.close();

  poly_index = -1;      //initialize

  for(int kt=1; kt<=Ntrials; kt++)                                 // for Ntrials trials
  {
    seconds1 = clock();                                                   // return
number of clock ticks at this time point

    for(int nt=1; nt<=kpoly; nt++)
    {
      poly_index++;              //increase the index, should be zero, initially
      if (poly_index >= npoly)
        poly_index -= npoly;            //ensure that it is less than the number used

      a = &(poly[poly_index].coeff[0]);                //set a to point to the ith polynomial
      dbk = poly[poly_index].deg;                                        //its degree
      poly_copy(dbk, a, dq, q);                                               //we need the
original polynomial for multiplication
```

```
//---------------Part 1---------------------//

square_ntimes(1, dq, q);                    // a_ans = q^2

poly_copy(dq, q, dtp, temp);                // temp = a_ans
square_ntimes(1, dq, q);                    //
poly_multiply(dq, q, dtp, temp);            // b = a_ans^2*a_ans

poly_copy(dq, q, dtp, temp);                // temp = b
square_ntimes(2, dq, q);                    //
poly_multiply(dq, q, dtp, temp);            // c = b^(2^2)*b

poly_copy(dq, q, dtp, temp);                // temp = c
square_ntimes(4, dq, q);                    //
poly_multiply(dq, q, dtp, temp);            //
poly_multiply(dq, q, dbk, a);          // d = c^(2^4)*c*a      = q^(2^9-1)

//---------------Part 2---------------------//

square_ntimes(1, dq, q);                    // e = d^2

poly_copy(dq, q, dtp, temp);                // temp = e
square_ntimes(9, dq, q);                    //
poly_multiply(dq, q, dtp, temp);            //
poly_multiply(dq, q, dbk, a);          // f = e^(2^9)*e*a      = q^(2^19-1)

//----------------Part 3---------------------//

square_ntimes(1, dq, q);                    // g = f^2

poly_copy(dq, q, dtp, temp);                // temp = g
square_ntimes(19, dq, q);                        //
poly_multiply(dq, q, dtp, temp);            // h = g^(2^19)*g

poly_copy(dq, q, dtp, temp);                // temp = h
square_ntimes(38, dq, q);                        //
poly_multiply(dq, q, dtp, temp);            //
poly_multiply(dq, q, dbk, a);          // i = h^(2^38)*h*a = q^(2^77-1)

//----------------Part 4--------------------//

square_ntimes(1, dq, q);                    // j = i^2

poly_copy(dq, q, dtp, temp);                // temp = j
square_ntimes(77, dq, q);                        // j^(2^77)
poly_multiply(dq, q, dtp, temp);            // k = j^(2^77)*j = q^(2^155-
2)
```

```
        //----------------End of one loop-----------//

      // if(kt==Ntrials)
      }

      seconds2 = clock();                                    // return number of clock ticks at this
time point
      cout << "CPU time : " << 1.0*(seconds2 - seconds1) << " (ms) for " << kpoly << "
loops." << endl;
      fout_cpu << 1.0*(seconds2 - seconds1) << endl;
   }

   fout_cpu.close();
   dq = poly_degree(154, q);
   poly_ofile(dbk, a, dq, q, fout_ans);                        // output last poly to file
   fout_ans.close();
   return 0;
}



void poly_read(int &d, int *p, ifstream &fin)
{
   int i;

   fin >> d;          //  read degree of p[]
   for (i=d; i>=0; i--)
      fin >> p[i];   //  read coefficients of p[]
}



void input(int &d, int *p)
{
   cout << "Input the degree of the inverse polynomial:  ";
   cin >> d;

    cout << "\nInput coefficients (must be 0 or 1): " << endl;
   for(int i=d; i>0; i--) {
      cout << "\nx^" << i << ": ";
      cin >> p[i];
   }
   cout << "\nconstant: ";
   cin >> p[0];

    for(int j=0; j<=d; j++) {
```

```
      if(p[j]!=0 && p[j]!=1) {
       cout << "Bad number detected" << endl;
       cout << "Program aborted..." << endl;
       exit(264);
     }
   }
}


void convert(int d, int *p, char *inverse)
{
   for(int i=0; i<=d; i++) {
      if(p[i]==0) inverse[i]=ch0;
         else inverse[i]=ch1;
   }
}


void square(int &d, char *inverse)                                    // return d and inverse
{
  char tempp[PMAX];
  init(PMAX-1, tempp);                                                         //
Initialization of tempp[]
   for(int i=0; i<=d; i++) {
      if(inverse[i]==ch1) tempp[2*i]=ch1;
  }
  d*=2;                                                       // new degree
  for(int j=0; j<=d; j++) inverse[j]=tempp[j];           //  Transfer tempp[] back to inverse[]
}


void square_ntimes(int n, int &d, char* a)
{
  for(int nt=1; nt<=n; nt++) {
    for(int i=d; i>=0; i--) {
       if(a[i]==ch1) a[2*i]=ch1;
      else a[2*i]=ch0;
      a[2*i+1]=ch0;                                       // add zero's in between
    }
    d*=2;                                        // new degree
    if(d>=DMOD) reduce2(d, a);
  }
}


void reduce(int &d, char *inverse)
{
```

```
  for(int i=DMOD; i<=d; i++) {
    if(inverse[i]==ch1) {
      inverse[i]=ch0;
      if(inverse[i-R1]==ch0) inverse[i-R1]=ch1;
      else inverse[i-R1]=ch0;
      if(inverse[i-R2]==ch0) inverse[i-R2]=ch1;
      else inverse[i-R2]=ch0;
    }
  }
  d=poly_degree(d, inverse);
}

//=================== new reduction method=========
void reduce2(int &d, char *inverse)
{
  int j1, j2, k1, k2;

  for(int i=DMOD; i<=d; i++) {
    if(inverse[i]==ch1) {
      inverse[i]=ch0;
      j1 = i-R1;
      j2 = i-R2;
      if(i<DMOD+R2) {
        if(inverse[j1]==ch0) inverse[j1]=ch1;
        else inverse[j1]=ch0;

        if(inverse[j2]==ch0) inverse[j2]=ch1;
        else inverse[j2]=ch0;
      } else {
        k1 = i-2*R2;
        k2 = i-R1-R2;
        if(inverse[j1]==ch0) inverse[j1]=ch1;
        else inverse[j1]=ch0;

        if(inverse[k1]==ch0) inverse[k1]=ch1;
        else inverse[k1]=ch0;

        if(inverse[k2]==ch0) inverse[k2]=ch1;
        else inverse[k2]=ch0;
      }
    }
  }
  d=poly_degree(DMOD, inverse);
}
```

```
void poly_multiply(int& da, char *a, int db, char *b)
{
  int i, j, k;
  int dc;
  char c[PMAX];
  int tm1, tm2;

  tm1 = clock();

  dc=da+db;
  init(dc, c);                                          //  initialization of c[]
  if((da==0 && a[0]==ch0) || (db==0 && b[0]==ch0)) return;        //  If both polynomials
are 0, the product will also be 0.

  for(i=da; i>=0; i--) {
    for(j=db; j>=0; j--) {
      k=i+j;
      if(a[i]==ch1 && b[j]==ch1) {
        if(c[k]==ch0) c[k]=ch1;
        else if(c[k]==ch1) c[k]=ch0;
        else {
          cout << "Invalid coefficients in poly_multiply; program aborted." << endl;
          exit(99);
        }
      }
    }
  }

  tm2 = clock();
  T_MULT += (tm2-tm1);

  if(dc>=DMOD) reduce2(dc, c);

  for(i=dc; i>=0; i--) a[i]=c[i];
  da=dc;


}


int poly_degree(int d, char *inverse)
{
  int i, j;

  for(i=d; i>=0; i--) {                         //  find out the degree of inverse[], which is j
    j=i;
    if(inverse[i]!=ch0) break;
```

```
  }
  return j;
}

void output(int d, char *inverse)
{
  if(d==0)
    cout << inverse[d] << endl;
  else {
    if(d==1 && inverse[0]==ch1) cout << "\nx + 1" << endl;
    else if(d==1 && inverse[0]==ch0) cout << "\nx" << endl;
    else {
      cout << "\nx^" << d;
      for(int i=d-1; i>=2; i--) {
        if(inverse[i]==ch1) cout << " + " << "x^" << i;
      }
      if(inverse[1]==ch1) cout << " + x";
      if(inverse[0]==ch1) cout << " + 1";
      cout << endl;
    }
  }
  cout << endl;
}


void poly_ofile(int dp, char* polyinput, int d, char *inverse, ofstream &fout)
{
  int k=0;                    //  k is a counter

  fout << "\nThe last polynomial used is: ";

  if(dp==0)
    fout << polyinput[dp] << endl;
  else {
    if(dp==1 && polyinput[0]==ch1) fout << "\nx + 1" << endl;
    else if(dp==1 && polyinput[0]==ch0) fout << "\nx" << endl;
    else {
      fout << "\n   x^" << dp; k++;
      for(int i=dp-1; i>=2; i--) {
        if(polyinput[i]==ch1) {

          if(i>=100)    fout << " + " << "x^" << i;
          else if(i>=10)          fout << " +  " << "x^" << i;
          else        fout << " +   " << "x^" << i;

          k++;
          if(k%10==0) fout << endl;
```

```
            }
        }

        if(polyinput[1]==ch1) {
            fout << " +     x";
            k++;
            if(k%10==0) fout << endl;
        }
        if(polyinput[0]==ch1) fout << " +     1";
        fout << endl;
    }
  }
  fout << "\nThe inverse for the last polynomial used is: ";

    if(d==0)
    fout << inverse[d] << endl;
  else {
    if(d==1 && inverse[0]==ch1) fout << "\nx + 1" << endl;
    else if(d==1 && inverse[0]==ch0) fout << "\nx" << endl;
    else {
      fout << "\n   x^" << d; k++;
      for(int i=d-1; i>=2; i--) {
        if(inverse[i]==ch1) {

          if(i>=100)    fout << " + " << "x^" << i;
          else if(i>=10)          fout << " +  " << "x^" << i;
          else        fout << " +   " << "x^" << i;

          k++;
          if(k%10==0) fout << endl;
        }
      }

      if(inverse[1]==ch1) {
        fout << " +     x";
        k++;
        if(k%10==0) fout << endl;
      }
      if(inverse[0]==ch1) fout << " +     1";
      fout << endl;
    }
  }
}


void init(int d, char *p)
{
```

```
  for(int i=0; i<=d; i++) p[i]=ch0;
}


void poly_copy(int ds, char *s, int &dt, char *t)
{
  for(int i=0; i<DMOD; i++)
    if(i<=ds) t[i]=s[i];            // copy array s[] to t[]
    else t[i]=ch0;

  dt=ds;
}
```

Lagrange's Theorem (Algorithm), Itoh's Technique, Byte Arrays, and Tables

```
/*
  Lagrange's Theorem using bytes and look-up tables
  8 Coefficients per Array Element
  Chen Zhao and Kristin Cordwell
  4-01-06
*/


#include <iostream>
#include <fstream>
#include <ctime>
using namespace std;


const int NBYTES=40;                                         //  number of
bytes for each polynomial
const int MAXCHAR=80;                                        //  maximum
length of filenames
const int MAX_NUM_POLYS = 2500;              //  read and use this number of
polynomials from the input file, up to MAX_NUM_POLYS
const int kpoly=2500;                                    //  actual number of
polynomials
const int Ntrials=50;                                 //  number of trials to average
cpu time

typedef struct
{
  unsigned char coeff[NBYTES];
  int  deg;
} poly_type;

typedef struct
{
  unsigned char low;
  unsigned char high;
} squarebytetype;

struct two_byte_type
{
  unsigned char byte[2];
};


squarebytetype squaretable[256];      //  table for squaring result
two_byte_type table[256][256];                  //  table for multiplying result
```

```
unsigned char low_93[256];                    //  modular reduction tables...
unsigned char high_93[256];
unsigned char low_155[256];
unsigned char high_155[256];
unsigned char spec_h93[32];
unsigned char spec_l93[32];
unsigned char spec_155[32];

poly_type poly[MAX_NUM_POLYS];                //for random polynomials


//        BEGIN DECLARATION OF FUNCTIONS
void poly_read(poly_type*, ifstream&);
void poly_copy(poly_type*, poly_type*);
void mult_table(two_byte_type t[][256]);
void poly_multiply(poly_type*, poly_type*);
void squarebyte(unsigned char, squarebytetype&);
void poly_square(int, poly_type*);
void reduce_table();
void poly_reduce(poly_type*);
void poly_output(poly_type*, ofstream&);
//        END DECLARATION OF FUNCTIONS

//=============================================================
==========
int main()
{
  ifstream fin;
  ofstream fout;
  ofstream fout_cpu;
  char inpfname [] = "poly_list.txt";         //  fixed input list of random
polynomials
  char outpfname [] = "poly_output.txt";    //  fixed output list of results
  char filename[MAXCHAR];
  int i;
  long seconds1;
  long seconds2;
  poly_type a[kpoly];                                          //  used to access list
of polynomials
  poly_type temp;
  poly_type backup[kpoly];

//precompute the byte multiplication table (for polynomials)
  mult_table(table);
  for(i=0;i<256;i++) squarebyte((unsigned char)i,squaretable[i]);

//precompute the modular reduction tables
```

```
   reduce_table();

   fout.open(outpfname);

   cout << "\nEnter output file name for CPU Times: ";                    // for CPU recording
   cin.getline(filename, MAXCHAR, '\n');
   fout_cpu.open(filename);
   fout_cpu << "CPU Time in ms: " << endl;
   cout << "\nCPU Time in ms: " << endl << endl;

//read in the random polynomials; we will find their inverses
   fin.open(inpfname);
   for(i=1; i<=kpoly; i++) {
      poly_read(&backup[i-1], fin);
      fout << "Input polynomial #" << i << " is: " << endl;
      poly_output(&backup[i-1], fout);
   }

   fout <<
"\n===============================================================
===================" << endl << endl;

   for(int trialcount=1; trialcount<=Ntrials; trialcount++)
   {
      for(i=0; i<kpoly; i++)
      {
         a[i].deg=backup[i].deg;
         poly_copy(&backup[i], &a[i]);
      }

      seconds1=clock();

      for(i=0; i<kpoly; i++)
      {
         //============Part 1============//

         poly_square(1, &a[i]);

         poly_copy(&a[i], &temp);
         poly_square(1, &a[i]);
         poly_multiply(&a[i], &temp);

         poly_copy(&a[i], &temp);
         poly_square(2, &a[i]);
         poly_multiply(&a[i], &temp);

         poly_copy(&a[i], &temp);
```

```
        poly_square(4, &a[i]);
        poly_multiply(&a[i], &temp);
        poly_multiply(&a[i], &backup[i]);

        //===========Part 2===========//

        poly_square(1, &a[i]);

        poly_copy(&a[i], &temp);
        poly_square(9, &a[i]);
        poly_multiply(&a[i], &temp);
        poly_multiply(&a[i], &backup[i]);

        ///===========Part 3===========//

        poly_square(1, &a[i]);

        poly_copy(&a[i], &temp);
        poly_square(19, &a[i]);
        poly_multiply(&a[i], &temp);

        poly_copy(&a[i], &temp);
        poly_square(38, &a[i]);
        poly_multiply(&a[i], &temp);
        poly_multiply(&a[i], &backup[i]);

        //===========Part 4===========//

        poly_square(1, &a[i]);

        poly_copy(&a[i], &temp);
        poly_square(77, &a[i]);
        poly_multiply(&a[i], &temp);

        //========End of one loop=======//
//      if(trialcount==Ntrials)
//      {
//        fout << "The inverse polynomial #" << i + 1 << " is: " << endl;
//        poly_output(&a[i], fout);       //  output answer
//      }

      }

      seconds2=clock();

//we only output the last value, after the loop is done, because we don't want
//to include the file I/O times in the time required to do the actual inverse calculation
```

```
        if(trialcount==Ntrials)
        {
          fout << "The last inverse polynomial #" << i << " is: " << endl;
          poly_output(&a[i-1], fout);    //  output answer        (decrement by 1 because i
exceeds the limit)
        }

      cout << 1.0*(seconds2-seconds1) << endl;
      fout_cpu << 1.0*(seconds2-seconds1) << endl;
    }

    fout_cpu.close();
    fin.close();
    return 0;
}

//------------------------------------------------------------------------------
void poly_read(poly_type* a, ifstream &fin)
//  Read degree and coefficients (int 1 or 0) of a polynomial from an input file
//  Convert int to bit (1/8 of a byte) and store in an array of unsigned char
//  a[] are the coefficients of the polynomial, a_deg is the degree of a[]
{
    int i;
    int coeff;
    int hi_byte;          //  highest index of bytes
    int hi_bit;           //  highest index of bit in hi_byte

    for (i=0; i<NBYTES; i++)
      (*a).coeff[i] = 0X00;                  //initialize a

    fin >> (*a).deg;                  //  read degree of the polynomial
    hi_byte=(*a).deg/8;
    hi_bit=(*a).deg%8;

    (*a).coeff[hi_byte]=0X00;                        //  initialize a.coeff[hi_byte]
    for (i=hi_bit; i>=0; i--) {               //  read the first partial byte
      fin >> coeff;
      if(coeff==1)
        (*a).coeff[hi_byte]=(*a).coeff[hi_byte]^(0X01<<i);
    }

    for (int ib=hi_byte-1; ib>=0; ib--) {           //  read the rest full bytes
      (*a).coeff[ib]=0X00;                                                   //  initialize this
(a[ib]) byte
      for (i=7; i>=0; i--) {                                         //  read 8 bits (a byte)
        fin >> coeff;
        if(coeff==1)
```

```
        (*a).coeff[ib] ^= (0X01<<i);
      }
    }
}

//-----------------------------------------------------------------------------
void poly_copy(poly_type* sender, poly_type* receiver)
//copies first polynomial onto the second polynomial
{
  for(int i=0; i<=19; i++) (*receiver).coeff[i]=(*sender).coeff[i];
  (*receiver).deg=(*sender).deg;
}

//-----------------------------------------------------------------------------
void mult_table(two_byte_type table[][256])
//construct table of multiplication elements, byte pairs
{
  int  i, j;
  unsigned char a, b;
  unsigned char product[2];

  for (i=0; i<256; i++) {
    a = (unsigned char) i;
    for (j=0; j<256; j++) {
      b = (unsigned char) j;
      product[0]=0X00; product[1]=0X00;              //  initialization of product
      for(int k=0; k<8; k++) {
        if(((b>>k) & 0X01) == 0X01) {
          product[0] ^= a << k;
          product[1] ^= a >> (8-k);
        }
      }
      table[i][j].byte[0] = product[0] & 0XFF;                  //low byte
      table[i][j].byte[1] = product[1] & 0XFF;                  //high byte
    }
  }

  //output_byte(table[5][5].byte[1]);
  //output_byte(table[5][5].byte[0]);

  return;
}

//-----------------------------------------------------------------------------
void poly_multiply(poly_type* a, poly_type *b)
{
  int ai;
```

```
  int bj;
  poly_type c;

  for(int i=0; i<NBYTES; i++)
    c.coeff[i]=0X00;
  //  initialize polynomial c
  c.deg=(*a).deg+(*b).deg;                                                    //  calculuate
degree of c

  for(i=0; i<=(*a).deg/8; i++) {
    ai = (int) (*a).coeff[i];
    for(int j=0; j<=(*b).deg/8; j++) {
      bj = (*b).coeff[j];
      c.coeff[i+j] ^= table[ai][bj].byte[0];
      c.coeff[i+j+1] ^= table[ai][bj].byte[1];
    }
  }
  while(c.deg>=155) poly_reduce(&c);
  poly_copy(&c, a);
}

//-----------------------------------------------------------------------------
void squarebyte(unsigned char a, squarebytetype& s)
{

  s.low = 0;
  s.high = 0;    //initialize

  s.low = s.low | (a&0x01);
  s.low = s.low | ((a&0x02)<<1);
  s.low = s.low | ((a&0x04)<<2);
  s.low = s.low | ((a&0x08)<<3);

  s.high = s.high | ((a&0x10)>>4);
  s.high = s.high | ((a&0x20)>>3);
  s.high = s.high | ((a&0x40)>>2);
  s.high = s.high | ((a&0x80)>>1);

  return;
}

//-----------------------------------------------------------------------------
void poly_square(int n, poly_type* a)
{
  for(int i=1; i<=n ; i++) {
    (*a).deg*=2;
    for(int j=19; j>=0; j--) {
```

```
        (*a).coeff[2*j+1]=(squaretable[int((*a).coeff[j])].high);
        (*a).coeff[2*j]=(squaretable[int((*a).coeff[j])].low);
      }
   while((*a).deg>=155) poly_reduce((a));
   }

}

//-----------------------------------------------------------------------------
void reduce_table()
{
   int i;
   //  create four tables used for reduction
   for(i=0; i<256; i++) low_93[i] = ((unsigned char) i) << 3;
   for(i=0; i<256; i++) high_93[i] = ((unsigned char) i) >> 5;
   for(i=0; i<256; i++) low_155[i] = ((unsigned char) i) << 5;
   for(i=0; i<256; i++) high_155[i] = ((unsigned char) i) >> 3;
   //  create special table for byte[19]
   for(i=0; i<32; i++) spec_h93[i] = (((unsigned char) i) >> 2) & 0X07;
   for(i=0; i<32; i++) spec_l93[i] = (((unsigned char) i) << 6) & 0XC0;
   for(i=0; i<32; i++) spec_155[i] = ((unsigned char) i) & 0X1F;
}

//-----------------------------------------------------------------------------
void poly_reduce(poly_type* a)
{
   int temp1, temp2;

   //  reduce bytes 20-39
   for(int i=39; i>=20; i--) {
      temp1=int((*a).coeff[i]);
      (*a).coeff[i-11]^=high_93[temp1];
      (*a).coeff[i-12]^=low_93[temp1];
      (*a).coeff[i-19]^=high_155[temp1];
      (*a).coeff[i-20]^=low_155[temp1];
      (*a).coeff[i]=0X00;              //  set coefficients of higher bytes to 0
   }

   //  reduce special byte 19
   temp2=int((*a).coeff[19]>>3);
   (*a).coeff[8]^=spec_h93[temp2];
   (*a).coeff[7]^=spec_l93[temp2];
   (*a).coeff[0]^=spec_155[temp2];
   (*a).coeff[19]&=0X07;              //  set bits 3-7 in byte 19 to 0

   for(int j=19; j>=0; j--) if((*a).coeff[j]!=0X00) break;
   for(int k=7; k>=0; k--) if((((*a).coeff[j]>>k) & 0X01) == 0X01) break;
```

```
  (*a).deg=8*j+k;
}

//---------------------------------------------------------------------------
void poly_output(poly_type* poly, ofstream &fout)
//  output one polynomial (poly) to screen and to file
{
  int hi_byte=(*poly).deg/8;
  int hi_bit=(*poly).deg%8;
  int count=1;
  int d;

  //cout << "   x^" << (*poly).deg;
  fout << "   x^" << (*poly).deg;
  for(int i=hi_bit-1; i>=0; i--) {
    if((((*poly).coeff[hi_byte]>>i)&0X01)==0X01)
    {
      //cout << " + x^" << 8*hi_byte+i;          //  output to screen
      fout << " + x^" << 8*hi_byte+i;            //  output to file
      count++;
    }
  }

  for(int j=hi_byte-1; j>=0; j--) {
    for(int k=7; k>=0; k--) {
      if((((*poly).coeff[j]>>k)&0X01)==0X01) {
        d=8*j+k;
        if(d==0) {
          //cout << "   + 1";
          fout << "   + 1";
        } else if(d==1) {
          //cout << " + x   ";
          fout << " + x   ";
        } else {
          if(d>=100) {
            //cout << " + x^" << d;
            fout << " + x^" << d;
          } else if (d>=10) {
            //cout << " + x^" << d << " ";
            fout << " + x^" << d << " ";
          } else {
            //cout << " + x^" << d << "  ";
            fout << " + x^" << d << "  ";
          }
        }
        count++;
        if(count%10==0) {
```

```
        //cout << endl;
        fout << endl;
      }
    }
  }
}
//cout << endl << endl;
fout << endl << endl;
}
```

Random Polynomial Generator

```
//poly_generator.cpp
//Chen Zhao and Kristin Cordwell
//Date:  15 March 2006
//
//This program generates random polynomials in GF(2^155)
//It generates a 1 or a 0 for each power of 2, from 2^154 down to 2^0
//and saves them in a file, along with the degree (the highest power)

#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <fstream.h>

ofstream fout;
char  filename[] = "poly_list3.txt";

const int highest_power = 154;

int  number_of_polys = 500;

//------------------------------------------------------------------------
void initialize_rand(void)
{
  /* Seed the random-number generator with current time so that
   * the numbers will be different every time we run.
   */
 srand( (unsigned)time( NULL ) );
  rand();
  return;
}

//------------------------------------------------------------------------
int  digit(void)
{
  int j;

  j = rand();

  return (j % 2);    //return a 1 or a 0
}

//=====================================================================
===========
void main(void)
```

```
{

int i;
int  j;
int  k;
int  degree = 0;
int flag = 0;

fout.open(filename);

for (j=0;j<number_of_polys;j++)
{
  flag = 0;                                            //initialize
  degree = 0;                                                  //initialize
  for (i=highest_power; i>=0; i--)
  {
    k = digit();
    if ((k==1) & (flag==0))                    //find when first coefficient of 1 occurs, the
degree
    {
      flag = 1;
      degree = i;
      fout << degree << endl;
    }
    if (flag==1)                                       //start when first 1 occurs
    {
      fout << k << ' ';
      if ((i % 10) == 0)
        fout << endl;           //new line, every 10 coefficients
    }
  }
}

}
```

Appendix D – Results Data Tables

EEA Bits (ms)					
# of loops	100	150	200	250	300
Trial 1	156	235	328	406	468
Trial 2	157	234	313	391	485
Trial 3	172	234	328	390	469
Trial 4	156	235	312	407	484
Trial 5	156	250	329	390	484
Trial 6	156	234	312	406	469
Trial 7	172	234	328	391	484
Trial 8	156	250	328	406	469
Trial 9	157	235	313	391	485
Trial 10	172	234	328	406	484
Trial 11	156	250	312	391	469
Trial 12	156	234	313	391	484
Trial 13	156	235	328	406	469
Trial 14	172	250	313	391	484
Trial 15	156	234	328	406	469
Trial 16	157	235	312	391	468
Trial 17	172	235	328	406	485
Trial 18	156	234	313	391	468
Trial 19	156	250	328	391	485
Trial 20	156	234	312	406	484
Trial 21	156	235	329	406	469
Trial 22	156	250	312	407	484
Trial 23	157	234	328	406	469
Trial 24	172	234	313	406	484
Trial 25	156	250	328	391	485
Trial 26	156	235	312	406	469
Trial 27	156	234	328	391	484
Trial 28	156	250	313	406	469
Trial 29	156	235	328	406	484
Trial 30	157	250	313	391	484
Trial 31	172	234	328	406	469
Trial 32	156	234	312	406	485
Trial 33	156	250	328	391	468
Trial 34	156	235	313	406	485
Trial 35	156	234	328	391	484
Trial 36	156	250	312	406	469
Trial 37	157	234	329	406	484
Trial 38	157	235	312	407	485
Trial 39	156	250	328	406	468
Trial 40	156	234	313	406	485
Trial 41	172	235	328	406	468
Trial 42	156	235	312	407	469
Trial 43	156	234	328	390	484
Trial 44	157	250	313	406	469
Trial 45	172	234	328	407	484
Trial 46	156	235	313	406	469
Trial 47	156	250	328	406	468
Trial 48	156	234	328	391	485
Trial 49	156	234	312	406	469
Trial 50	156	234	328	391	484
Average	159.04	238.76	320.62	400.34	477.16
Time per polynomial	1.5904	1.591733	1.6031	1.60136	1.590533

EEA Bytes (ms)						
# of loops	500	1000	1500	2000	2500	3000
Trial 1	63	125	188	218	281	344
Trial 2	62	109	172	219	281	359
Trial 3	47	125	172	234	297	344
Trial 4	63	109	171	218	281	344
Trial 5	62	125	188	235	297	359
Trial 6	63	110	172	234	297	344
Trial 7	62	125	172	235	281	359
Trial 8	63	109	172	234	297	344
Trial 9	62	125	187	234	297	344
Trial 10	63	109	172	235	281	359
Trial 11	47	125	172	234	297	344
Trial 12	62	110	172	234	297	344
Trial 13	63	125	172	235	281	359
Trial 14	62	109	187	234	297	344
Trial 15	47	125	172	235	297	359
Trial 16	62	110	172	234	281	344
Trial 17	63	125	172	234	297	344
Trial 18	62	109	172	219	297	344
Trial 19	47	125	187	218	281	344
Trial 20	47	125	172	235	297	359
Trial 21	62	109	172	234	281	344
Trial 22	63	125	172	235	281	344
Trial 23	62	110	187	234	297	359
Trial 24	63	125	172	234	297	344
Trial 25	47	109	172	235	297	343
Trial 26	62	125	172	234	281	360
Trial 27	63	109	187	234	282	344
Trial 28	62	125	172	235	297	359
Trial 29	47	110	172	234	296	344
Trial 30	63	125	172	235	297	343
Trial 31	62	109	187	234	282	344
Trial 32	63	125	172	234	296	344
Trial 33	62	110	172	235	297	359
Trial 34	47	125	172	234	282	344
Trial 35	46	109	172	234	296	343
Trial 36	63	125	187	235	297	360
Trial 37	62	125	172	234	282	344
Trial 38	63	109	172	235	296	343
Trial 39	62	125	172	234	297	360
Trial 40	63	110	188	219	282	343
Trial 41	62	125	171	235	281	360
Trial 42	63	109	172	218	297	344
Trial 43	47	125	188	219	297	343
Trial 44	47	109	172	234	281	360
Trial 45	63	125	172	235	297	343
Trial 46	62	110	171	234	297	360
Trial 47	63	125	172	234	281	344
Trial 48	47	109	188	235	297	343
Trial 49	62	125	172	234	297	360
Trial 50	63	110	172	234	281	343
Average	58.76	117.5	175.64	231.84	290.62	349.06
Time per polynomial	0.11752	0.1175	0.117093	0.11592	0.116248	0.116353

Method	Extended Euclidean Algorithm Bytes (ms)					
# of loops	500	1000	1500	2000	2500	3000
Trial 1	63	109	172	250	282	359
Trial 2	62	125	188	218	282	344
Trial 3	63	110	171	219	296	359
Trial 4	62	110	172	234	282	344
Trial 5	47	125	172	234	297	344
Trial 6	63	109	172	235	296	359
Trial 7	62	125	172	234	282	344
Trial 8	63	109	172	235	297	344
Trial 9	62	125	172	234	296	344
Trial 10	47	110	187	234	282	359
Trial 11	63	125	188	235	297	344
Trial 12	62	125	172	234	281	359
Trial 13	63	109	187	234	282	344
Trial 14	46	125	172	235	297	344
Trial 15	47	125	172	234	296	359
Trial 16	62	110	188	235	282	344
Trial 17	63	110	171	234	297	359
Trial 18	62	125	188	234	281	344
Trial 19	47	109	172	235	282	359
Trial 20	47	125	172	234	297	344
Trial 21	62	125	172	234	281	344
Trial 22	63	109	172	235	297	359
Trial 23	62	109	172	234	297	344
Trial 24	47	125	171	235	281	359
Trial 25	63	110	172	234	297	344
Trial 26	62	125	172	234	297	344
Trial 27	63	125	188	235	281	343
Trial 28	62	109	172	234	297	344
Trial 29	63	125	171	234	297	343
Trial 30	47	125	172	235	281	344
Trial 31	47	109	188	234	297	360
Trial 32	63	125	172	235	281	343
Trial 33	62	125	172	234	281	360
Trial 34	63	110	187	234	297	343
Trial 35	46	125	172	235	281	360
Trial 36	63	125	172	234	297	344
Trial 37	62	109	172	234	297	343
Trial 38	63	125	187	235	281	344
Trial 39	62	110	172	234	297	343
Trial 40	47	125	172	235	297	344
Trial 41	63	109	187	234	281	344
Trial 42	62	125	172	234	297	359
Trial 43	63	109	172	235	297	344
Trial 44	62	125	172	234	281	359
Trial 45	63	110	172	234	297	359
Trial 46	47	125	171	235	297	360
Trial 47	62	109	172	219	281	344
Trial 48	63	125	172	235	297	344
Trial 49	62	109	188	234	297	344
Trial 50	47	109	172	234	297	359
Average	58.44	117.5	175.64	233.74	290.36	349.36
per polynomial	0.11688	0.1175	0.117093	0.11687	0.116144	0.116453

	Russian Peasant Method (1000 ms)
# of Loops	100
Trial 1	3.141
Trial 2	3.172
Trial 3	3.156
Trial 4	3.203
Trial 5	3.156
Trial 6	3.187
Trial 7	3.188
Trial 8	3.156
Trial 9	3.141
Trial 10	3.188
Trial 11	3.203
Trial 12	3.157
Trial 13	3.141
Trial 14	3.156
Trial 15	3.156
Trial 16	3.156
Trial 17	3.157
Trial 18	3.187
Trial 19	3.156
Trial 20	3.141
Trial 21	3.156
Average	3.16447619
Time per polynomial	0.031644762

LGT - Itoh's Method in Bits (ms)					
# of loops	50	100	150	200	250
Trial 1	125	266	406	515	656
Trial 2	141	265	391	532	656
Trial 3	125	250	391	531	657
Trial 4	141	266	406	515	656
Trial 5	125	266	390	532	656
Trial 6	125	265	407	531	656
Trial 7	125	266	390	516	657
Trial 8	125	266	407	531	656
Trial 9	141	265	390	531	641
Trial 10	125	266	391	516	672
Trial 11	141	250	406	531	656
Trial 12	125	265	391	516	656
Trial 13	140	266	390	531	657
Trial 14	125	266	407	531	656
Trial 15	141	265	406	516	672
Trial 16	125	266	390	531	656
Trial 17	125	266	407	531	656
Trial 18	140	265	390	516	657
Trial 19	125	250	407	531	656
Trial 20	141	266	390	531	656
Trial 21	125	265	406	516	656
Trial 22	141	266	391	531	657
Trial 23	125	266	390	531	656
Trial 24	125	265	391	532	656
Trial 25	140	250	406	515	656
Trial 26	125	266	391	532	657
Trial 27	141	266	406	515	656
Trial 28	125	265	391	515	656
Trial 29	141	266	390	532	656
Trial 30	125	265	407	531	657
Trial 31	140	250	390	531	656
Trial 32	125	266	406	516	656
Trial 33	125	266	391	531	656
Trial 34	141	265	391	516	657
Trial 35	125	266	391	516	656
Trial 36	140	250	391	531	656
Trial 37	125	250	406	516	656
Trial 38	141	265	390	531	657
Trial 39	125	266	407	531	656
Trial 40	141	265	390	532	656
Trial 41	125	266	391	531	656
Trial 42	125	266	391	531	657
Trial 43	140	250	406	516	656
Trial 44	125	250	390	516	656
Trial 45	141	266	391	531	656
Trial 46	125	266	406	531	657
Trial 47	141	265	391		656
Trial 48	125	266	391		656
Trial 49	125	250	390		656
Trial 50	141	265	406		657
Average	132.6	264.5	399.56	529.8261	661.58
Time per polynomial	2.652	2.645	2.663733	2.64913	2.64632

Method	Lagrange Bytes (ms)					
# of loops	500	1000	1500	2000	2500	3000
Trial 1	78	141	219	297	375	453
Trial 2	78	156	235	313	391	454
Trial 3	78	156	218	297	375	453
Trial 4	78	157	235	312	375	453
Trial 5	63	140	218	297	375	453
Trial 6	78	157	235	297	390	453
Trial 7	78	156	219	312	375	453
Trial 8	78	140	234	297	375	469
Trial 9	78	157	234	313	375	453
Trial 10	63	156	219	297	391	453
Trial 11	78	156	234	296	375	453
Trial 12	78	141	219	313	375	453
Trial 13	78	156	235	297	375	454
Trial 14	78	156	218	312	391	453
Trial 15	78	141	235	297	375	453
Trial 16	78	156	218	297	375	453
Trial 17	63	156	235	313	375	453
Trial 18	78	157	219	296	390	453
Trial 19	78	140	234	297	375	453
Trial 20	78	157	219	297	375	453
Trial 21	78	156	219	297	375	469
Trial 22	78	140	234	312	375	453
Trial 23	63	157	219	297	391	453
Trial 24	78	156	235	297	375	454
Trial 25	78	156	218	312	375	453
Trial 26	78	156	235	297	391	453
Trial 27	78	141	218	313	375	453
Trial 28	79	156	235	297	375	453
Trial 29	78	157	219	297	375	453
Trial 30	78	156	234	312	390	453
Trial 31	62	140	234	297	375	453
Trial 32	78	157	219	312	375	453
Trial 33	79	156	234	297	375	453
Trial 34	78	156	219	297	391	453
Trial 35	78	156	235	313	375	454
Trial 36	78	157	218	297	375	453
Trial 37	62	156	235	296	375	453
Trial 38	79	156	218	297	375	453
Trial 39	78	156	235	297	390	453
Trial 40	78	157	234	312	375	453
Trial 41	78	156	219	297	375	453
Trial 42	78	141	234	297	391	453
Trial 43	78	156	219	313	375	454
Trial 44	63	156	234	296	375	453
Trial 45	78	156	219	313	375	453
Trial 46	78	157	235	297	391	469
Trial 47	78	156	234	297	375	453
Trial 48	78	156	219	312	375	453
Trial 49	78	156	234	297	375	453
Trial 50	63	141	219	297	390	453
Average	75.62	152.82	226.9	302.18	379.06	454.06
per polynomial	0.15124	0.15282	0.151267	0.15109	0.151624	0.151353

```
(* Input Polynomial *)
a =
x^151+x^150+x^149+x^147+x^146+x^145+x^140+x^139+x^137+x^136+
x^135+x^133+x^132+x^130+x^129+x^124+x^123+x^122+x^121+x^119+
x^118+x^117+x^109+x^108+x^105+x^102+x^101+x^100+x^99+x^94+x^
91+x^90+x^88+x^87+x^86+x^85+x^84+x^83+x^80+x^76+x^74+x^73+x^
70+x^69+x^67+x^66+x^65+x^63+x^59+x^58+x^56+x^54+x^53+x^52+x^
51+x^48+x^41+x^38+x^37+x^36+x^34+x^31+x^30+x^29+x^27+x^26+x^
25+x^22+x^21+x^20+x^19+x^15+x^13+x^12+x^10+x^9+x^8+x^6+x^4+x
^3+x
```

$$x + x^3 + x^4 + x^6 + x^8 + x^9 + x^{10} + x^{12} + x^{13} + x^{15} + x^{19} + x^{20} + x^{21} + x^{22} + x^{25} + x^{26} + x^{27} + x^{29} + x^{30} + x^{31} + x^{34} + x^{36} + x^{37} + x^{38} + x^{41} + x^{48} + x^{51} + x^{52} + x^{53} + x^{54} + x^{56} + x^{58} + x^{59} + x^{63} + x^{65} + x^{66} + x^{67} + x^{69} + x^{70} + x^{73} + x^{74} + x^{76} + x^{80} + x^{83} + x^{84} + x^{85} + x^{86} + x^{87} + x^{88} + x^{90} + x^{91} + x^{94} + x^{99} + x^{100} + x^{101} + x^{102} + x^{105} + x^{108} + x^{109} + x^{117} + x^{118} + x^{119} + x^{121} + x^{122} + x^{123} + x^{124} + x^{129} + x^{130} + x^{132} + x^{133} + x^{135} + x^{136} + x^{137} + x^{139} + x^{140} + x^{145} + x^{146} + x^{147} + x^{149} + x^{150} + x^{151}$$

```
(* Input Calculated Inverse from Program *)
ainv =
x^154+x^150+x^148+x^147+x^146+x^143+x^141+x^140+x^138+x^137+
x^136+x^133+x^132+x^130+x^128+x^124+x^123+x^119+x^115+x^113+
x^111+x^109+x^108+x^101+x^100+x^97+x^96+x^93+x^90+x^89+x^87+
x^86+x^84+x^83+x^80+x^77+x^74+x^65+x^64+x^61+x^60+x^58+x^55+
x^54+x^53+x^51+x^49+x^46+x^44+x^43+x^42+x^41+x^40+x^37+x^36+
x^35+x^34+x^33+x^30+x^29+x^25+x^23+x^22+x^20+x^16+x^15+x^14+
x^13+x^12+x^9+x^7+x^6+x^4+1
```

$$1 + x^4 + x^6 + x^7 + x^9 + x^{12} + x^{13} + x^{14} + x^{15} + x^{16} + x^{20} + x^{22} + x^{23} + x^{25} + x^{29} + x^{30} + x^{33} + x^{34} + x^{35} + x^{36} + x^{37} + x^{40} + x^{41} + x^{42} + x^{43} + x^{44} + x^{46} + x^{49} + x^{51} + x^{53} + x^{54} + x^{55} + x^{58} + x^{60} + x^{61} + x^{64} + x^{65} + x^{74} + x^{77} + x^{80} + x^{83} + x^{84} + x^{86} + x^{87} + x^{89} + x^{90} + x^{93} + x^{96} + x^{97} + x^{100} + x^{101} + x^{108} + x^{109} + x^{111} + x^{113} + x^{115} + x^{119} + x^{123} + x^{124} + x^{128} + x^{130} + x^{132} + x^{133} + x^{136} + x^{137} + x^{138} + x^{140} + x^{141} + x^{143} + x^{146} + x^{147} + x^{148} + x^{150} + x^{154}$$

```
(* Check to see if the inverse really works! *)
PolynomialMod[a * ainv, x^155 + x^62 + 1, Modulus→2]
```

1