Elliptic Curve Computations

New Mexico Supercomputing Challenge

Final Report

April 1st, 2009

Team 66

Manzano High School

Team Members

Kristin Cordwell

Chen Zhao

Teacher

Stephen Schum

Project Mentor

William Cordwell

Executive Summary

Elliptic curves are increasingly being used in cryptographic applications, particularly in public key exchange and in authentication, where the Federal Government has approved the Elliptic Curve Digital Signature Algorithm. Because of the rising prevalence of elliptic curve cryptography, an important area of study is how to conduct the necessary elliptic curve group operations in the most efficient manner possible.

Code was written to handle large integer arithmetic and to perform the basic operations of elliptic curve point addition and point doubling. Building on this underlying structure, we investigated the principal operation for elliptic curve calculations–point multiplication modulo a large prime p. Point multiplication requires the calculation of inverses modulo p, and, as finding inverses is a major driver for the computational cost of elliptic curve operations, we gave some extra attention to finding an efficient scheme for these calculations. To this end, we used a modified Extended Euclidean Algorithm (rather than Lagrange's Theorem) to find our inverses, implementing a trick known as the almost inverse algorithm. In addition to regular affine and projective representations, as the major focus of our project we implemented and tested for the first time a new technique invented by Rich Schroeppel that is designed to greatly speed up point multiplication.

With approximately seven inverses, Schroeppel's method, taken in conjunction with the almost inverse algorithm, is as fast as projective, which only requires one inverse. Both take around 33-35 milliseconds to multiply a point by a 256-bit number. We estimate that with more emphasis on optimizing reciprocal calculations, Schroeppel's method could be about 20% faster than projective coordinates for point multiplication in our implementation.

Contents

List of Figures

Introduction

Background: The Advantages of Elliptic Curves

Public key cryptography, invented in the 1970s, primarily has used two schemes, RSA, named after Rivest, Shamir, and Adelman, and Diffie-Hellman key exchange. RSA has been extremely popular, both for encrypting small amounts of data and for generating digital signatures. It is, however, computationally intensive, using lots of modular exponentiations, so, for efficiency, one would like to keep the parameters small. RSA's security, however, is based on the difficulty of factoring large composite numbers, so one would like the parameters to be large for this reason.

In symmetric key (shared secret key) systems the security is usually measured in the number of tries for an exhaustive search. A decade ago, cryptographers built a special purpose computer to break DES (Digital Encryption Standard), which has a key length of 56 bits, for a "strength" of 2^{56}. Each extra bit of key length adds a multiple of two in the difficulty of breaking the system, so an extra 10 bits of key length gives a multiple of more than 1000 in extra strength against an attacker.

For RSA, however, the complexity of factoring is less difficult than exhaustive search. The General Number Field Sieve, currently the most efficient method for factoring large, general numbers, has complexity (roughly, the amount of work required to run an algorithm) of $f(n) = O(e^{c\cdot[ln(n)]^{\frac{1}{3}}[ln(ln(n))]^{\frac{2}{3}}})$, where n is the number to be factored. A good value for c is 1.923, so we can use the formula to find the increase in security with an increase in the size of n. People have factored a general 663-bit number ($n \approx 2^{663}$), and a 1024-bit number should show an increase in strength of about $\frac{f(2^{1024})}{f(2^{663})} \approx 34000$. Going from 1024 bits to 2048 bits gives an increase in strength of about 1.2 billion, or a little more than 2^{30}.

The table below shows a reasonably accurate comparison for the security of symmetric key

block cipher systems, RSA, and elliptic curves over a prime p. The RSA modulus sizes are ones that are multiples of 1024 and most closely match the security of the symmetric key sizes.

Security (bits)	80	112 (Triple-DES)	128 (AES-low)	256 (AES-high)
RSA modulus n	1024	2048	3072	15360
Elliptic Curve p	160	224	256	512

DES = Digital Encryption Standard

AES = Advanced Encryption Standard

Although the group operation is somewhat more complicated than exponentiation, the much smaller key sizes make elliptic curves much more efficient than RSA.

Federal Standards

The National Security Agency (NSA) has approved Suite B cryptography for protecting both classified and unclassified national security information [15]. (Suite A contains classified algorithms.) Suite B comprises the following algorithms:

Encryption	Advanced Encryption Standard (AES) - FIPS 197 (with key sizes of 128 and 256 bits)
Digital Signature	Elliptic Curve Digital Signature Algorithm - FIPS 186-2 (using the curves with 256 and 384-bit prime moduli)
Key Exchange	Elliptic Curve Diffie-Hellman - Draft NIST Special Pub. 800-56 (using the curves with 256 and 384-bit prime moduli)
Hashing	Secure Hash Algorithm - FIPS 180-2 (using SHA-256 and SHA-384)

In this project, we focus on the NIST FIPS 186-2 elliptic curve with 256-bit prime modulus [13].

Basic Elliptic Curves

Equations and Points

An elliptic curve is a graph of an equation that is quadratic in y and cubic in x. The most general equation that we will consider is of the form $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$, where $a_1, \ldots, a_6$ are constants in the underlying field. If the field is not characteristic 2 (essentially, mod 2, see Appendix A), then we can reduce the form of the equation by noticing that $(y + \frac{a_1x}{2} + \frac{a_3}{2})^2 = x^3 + (a_2 + \frac{{a_1}^2}{4})x^2 + (a_4 + \frac{a_1a_3}{2})x + (\frac{{a_3}^2}{4} + a_6)$. By setting $y_1 = y + \frac{a_1x}{2} + \frac{a_3}{2}$, we can rewrite the original equation as ${y_1}^2 = x^3 + \hat{a_2}x^2 + \hat{a_4}x + \hat{a_6}$, with appropriate choices of $\hat{a_2}, \hat{a_4}, \hat{a_6}$. If the characteristic is neither 2 nor 3, we can eliminate the x^2 term by letting $x = x_1 - \frac{\hat{a_2}}{3}$ to obtain ${y_1}^2 = {x_1}^3 + Ax_1 + B$, for the appropriate constants A and B. Although we may consider a field of characteristic 2 later, for the most part we will focus on equations of this form, which we will simply write as $y^2 = x^3 + Ax + B$.

Figures 1 and 2 (below) show typical elliptic curves over the real numbers. When going from the equation linear in y to the elliptic curve (y^2), notice that the negative portion in y of the linear curve (red) is discarded. Setting the positive portion equal to y^2 gives a reflection about the x-axis of the original positive portion and causes some slight smoothing of the curve as well.

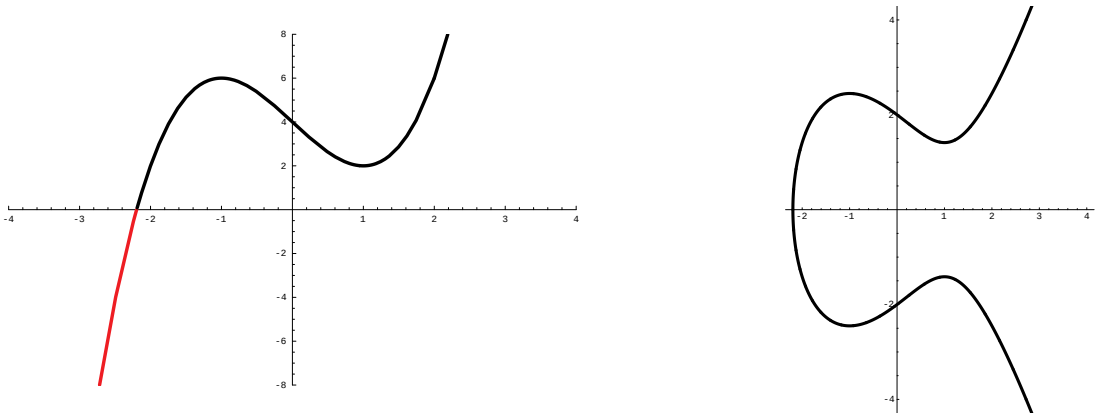

Figure 1: $y = x^3 - 3x + 4$ and $y^2 = x^3 - 3x + 4$

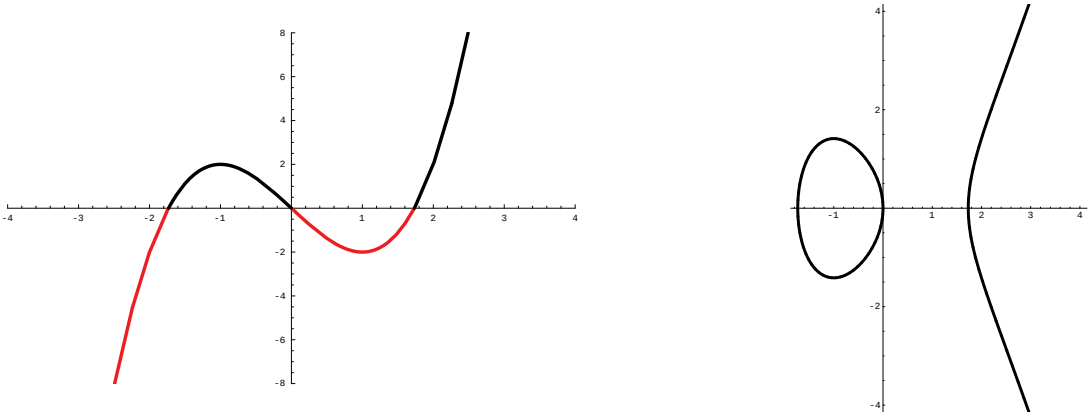

Figure 2: $y = x^3 - 3x$ and $y^2 = x^3 - 3x$

The shape of the elliptic curve is largely determined by the behavior of the cubic in x. Over the real numbers, the regular cubic equation $y = x^3 + Ax + B$ always has at least one real root and could have three real roots, possibly including duplicate roots. The equation $y = x^3$, for example, has a triple root at $x = 0$, while $y = x^3 - 3x + 2 = (x - 1)^2(x + 2)$ has a double root at $x = 1$ as well as a single root at $x = -2$.

We need to be able to take a tangent line at every point of the elliptic curve, which means that the graph must be smooth everywhere. When we extend the cubic equation to the elliptic curve equation, for the case of three repeated roots ($y^2 = x^3$), we get a cusp at $x = 0$, as seen in Figure 3, so no tangent exists at that point. Similarly, in the case of a cubic with a double root, for example $y^3 = x^3 - 3x + 2$, we get a crossing at the double root, so, again, we cannot find a unique tangent at that point (Figure 4). For this reason, we require our elliptic curves to have three distinct roots. Equivalently, we can require that the discriminant $((r_1 - r_2)(r_1 - r_3)(r_2 - r_3))^2 \neq 0$, where r_1, r_2, and r_3 are the three roots of the cubic equation. In terms of the coefficients, $4A^3 + 27B^2 \neq 0$.

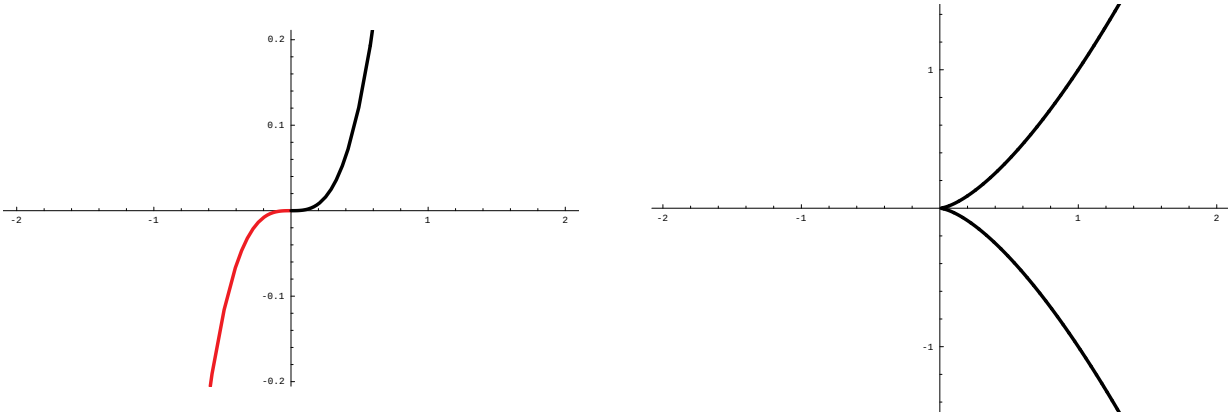

Figure 3: Three repeated roots

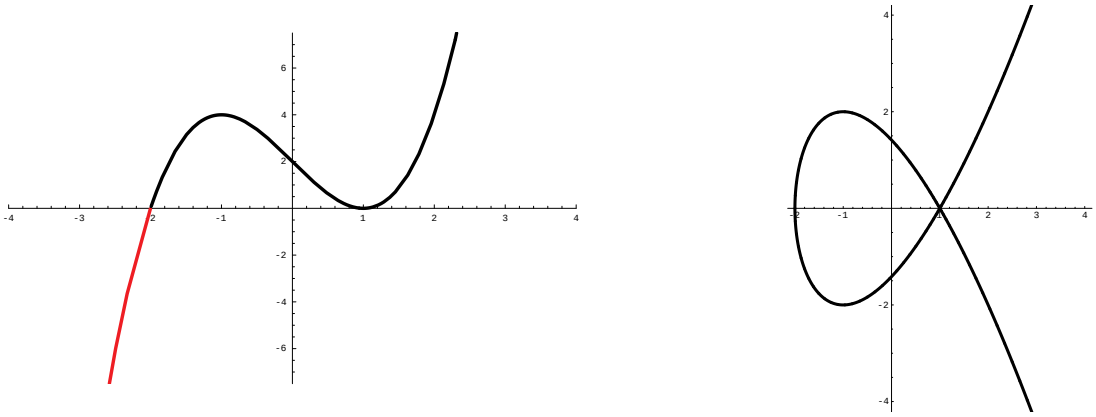

Figure 4: Two repeated roots

Group Addition Law

Affine Coordinates

A point on an elliptic curve is the ordered pair (x, y), where the coordinates satisfy the elliptic curve equation. Let P and Q be points on an elliptic curve. Then we can add two points geometrically by taking the line that joins them, finding where that line intersects the curve in a third point, and then reflecting that point about the x-axis, which is also on the curve. To add a point to itself, we take the tangent to the curve at that point, and do the same process (recall that elliptic curves must have well-defined tangent lines at every point). It is shown in [11] that the points of an elliptic curve, together with this addition operation, form a group.

Arithmetically, if P and Q are distinct points on the curve, where $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, we will want to find $P + Q = N$. Letting the slope of the line through P and Q be $m = \frac{y_2 - y_1}{x_2 - x_1}$, we find that the equation of the line is $y = m(x - x_1) + y_1$. Plugging in for y in the equation of the curve gives an equation of x for the third point where the line intersects the curve (it already intersects at P and Q),

$(m(x - x_1) + y_1)^2 = x^3 + Ax + B$.

Substituting gives $0 = x^3 - m^2x^2 + m^2x_1^2 + (A + 2m^2x_1 - 2my_1)x + (B + 2mx_1y_1 - y_1^2)$.

Since we know that two roots of the equation are x_1 and x_2, we have

$(x - x_1)(x - x_2)(x - x_t) = x^3 - m^2x^2 + m^2x_1^2 + (A + 2m^2x_1 - 2my_1)x + (B + 2mx_1y_1 - y_1^2)$,

where x_t is the third root.

Expanding the left side, and setting the coefficients of the x^2 term equal gives $x_3 = x_t = m^2 - x_1 - x_2$, and thus $y_t = m(x_3 - x_1) + y_1$.

Finally, reflecting about the x-axis changes the sign of y_t to give $P + Q = N = (x_3, y_3)$, where $x_3 = m^2 - x_1 - x_2$ and $y_3 = m(x_1 - x_3) - y_1$. If Q is P reflected about the x-axis, then

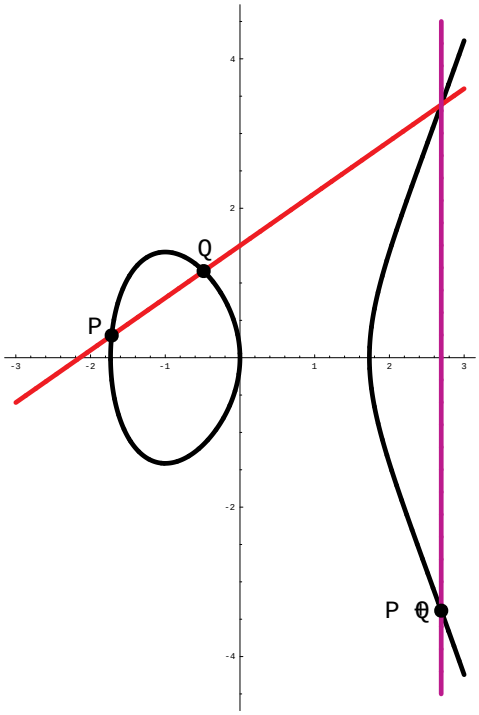

Figure 5: Point Addition

the line is vertical, and we say $P + Q = O$, the point at infinity, which is defined to be the identity element of the group. This also means that the reflection of P about the x-axis is $-P$.

We notice that if P and Q are close together, the line through them approaches the tangent line. Therefore, when $P = Q = (x_1, y_1)$, we take L to be the tangent line through the point, and we are finding $P + P = 2P$. We find the slope m of L through implicit differentiation: $2y\frac{dy}{dx} = 3x^2 + A$, so $m = \frac{dy}{dx} = \frac{3{x_1}^2+A}{2y_1}$.

If $y_1 = 0$, then L is vertical, so we set $P + P = 2P = O$.

For $y_1 \neq 0$, we have for the equation of line L,

$y = m(x - x_1) + y_1$. From this, we get the cubic equation $0 = x^3 - m^2x^2 + \cdots$. We know that x_1 is a double root, so, in a similar manner to as before, we can find $x_3 = m^2 - 2x_1$ and $y_3 = m(x_1 - x_3) - y_1$. This is called *point doubling*.

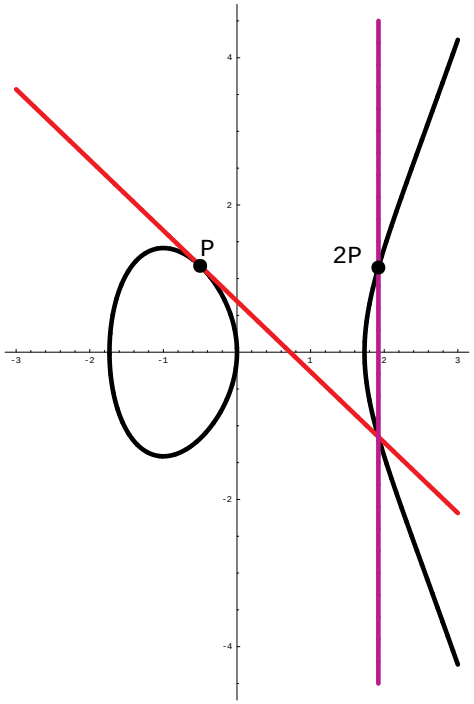

Figure 6: Point Doubling

Finally, for the case in which $Q = O$, we know that the line joining P and Q is vertical, and it intersects our curve at $-P$, where $-P$ is the reflection of P across the x-axis. Thus, $P + O = P$.

Although affine is the most straightforward way to add and double points, each addition (or doubling) requires the calculation of an inverse mod p. Just as long division takes far longer than multiplication or addition, so the calculation of inverses (reciprocals) is much more computationally expensive than multiplications or additions. Hence, we would expect that using affine coordinates might take more time than an alternate method that utilizes fewer reciprocal calculations.

Projective Coordinates

We can also express points in projective coordinates, using x, y, and z to describe the location in projective space. Using projective rather than affine coordinates allows us to avoid finding reciprocals, a computationally expensive operation. Somewhat working against this is the fact that it requires an additional coordinate, so one must perform extra multiplications in point additions and doublings.

We express our two points to be added as $P = (x_1, y_1, z_1)$ and $Q = (x_2, y_2, z_2)$, and we call our elliptic curve $y^2z = x^3 + Axz^2 + Bz^3$. Then we have $(x_1, y_1, z_1) + (x_2, y_2, z_2) = (x_3, y_3, z_3)$, and we compute x_3, y_3, and z_3 for each of the following cases.

If $P \neq \pm Q$, we have

$u = y_2z_1 - y_1z_2$, $v = x_2z_1 - x_1z_2$, $w = u^2z_1z_2 - v^3 - 2v^2x_1z_2$, and we find that

$x_3 = vw$,

$y_3 = u(v^2x_1z_2 - w) - v^3y_1z_2$,

$z_3 = v^3z_1z_2$.

If $P = Q$, we obtain $t = Az_1{}^2 + 3x_1{}^2$, $u = y_1z_1$, $v = ux_1y_1$, $w = t^2 - 8v$, yielding

$x_3 = 2uw$,

$y_3 = t(4v - w) - 8y_1{}^2u^2$,

$z_3 = 8u^3$.

Finally, for $P = -Q$, we have $P + Q = O$.

Elliptic Curves Over Finite Fields

Structure of the Elliptic Curve

The previous discussion pertained to elliptic curves over the real numbers. So, for example, all of the x and y coordinates were real numbers, and there were infinitely many solutions to the elliptic curve equation. Most modern cryptographic applications utilize a finite group, and we can achieve this with the elliptic curves by letting the underlying field be a finite field, $GF(p^n)$. Typically, people use either $GF(p)$, the integers modulo a large prime p, or $GF(2^n)$. We will primarily focus on $GF(p)$.

Since there are only a finite number of possible x coordinates and y coordinates, the number of possible solutions to an elliptic curve equation over a finite field is necessarily finite. An elliptic curve over a finite field, then, is the finite number of points that satisfy the elliptic curve equation together with the group operation, where we can add points together. The number of points in the curve $E(GF(p))$, is always equal to $p + 1 - a$, where $|a| \leq 2\sqrt{p}$ [11]. It is straightforward to compute the number of points using various algorithms. The formulas for point addition and point doubling, in both affine and projective representations, are exactly the same as before, except that we must substitute multiplying by an inverse element of the field, rather than division. For example, in affine point doubling, instead of $m = \frac{3x_1^2+A}{2y_1}$, we have $m = (3x_1^2 + A)((2y_1)^{-1})$. Inverses in finite fields are readily found using

either the Extended Euclidean Algorithm or Lagrange's Theorem. Just as long division is much more computationally expensive than multiplication in regular arithmetic, so finding inverses in finite fields is far more computationally expensive than multiplications in the field. Much of our work focusses on reducing the number of calculations of inverses, so as to minimize the computations involved and speed up the calculations.

Elliptic Curve $P-256$

The elliptic curve $P-256$ is a NIST-specified curve over a 256-bit prime [13]. It is also one of two curves specified in NSA's Suite B algorithms [15]. The equation of the curve is $y^2 = x^3 - 3x + b \pmod{p}$, where

$b = 5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b_{16}$

$p = 115792089210356248762697446949407573530086143415290314195533631308867097853951$,

for p the prime modulus of the underlying field.

$r = 115792089210356248762697446949407573529996955224135760342422259061068512044369$,

for r the number of points on the curve.

The chosen base point G has coordinates (G_x, G_y), where

$G_x = 6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296_{16}$

$G_y = 4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5_{16}$

We use this curve throughout our code.

Point Multiplication on Elliptic Curves

Many of the cryptographic applications of elliptic curves consider a fixed base point, G, of an elliptic curve, and compute multiples nG of G, where nG is defined to be iterated addition of n copies of G. Typically, n is approximately the size of the group; or, over $GF(p)$, this is approximately p. It is impractical for common values of p (a 100 to 300-bit number) to compute nG by iterated addition. Instead, we use a variant of the Russian Peasant Method [4] in which the coefficient n is converted to binary and the following algorithm is applied:

For the high bit (necessarily a 1), load the answer register R with G. Point to the next lower bit.

Loop:

$R \leftarrow 2R$;

If the bit is a 1, $R \leftarrow R + G$;

If the bit is the last bit, exit loop;

Return R.

For example, consider computing $13G$.

$13_{10} = 1101_2$

1101	G	= G
1101	2G + G	= 3G
1101	2(3G)	= 6G
1101	2(6G) + G	= 13G

For a typical p-bit number, there will be $\frac{p}{2}$ zeros and $\frac{p}{2}$ ones, so there will be about p point doublings and $\frac{p}{2}$ point additions. Each of these involves computing a reciprocal (inverse).

Efficient Reciprocal Calculation

Just as long division is far more computationally intensive than multiplication, the calculation of inverses (reciprocals) mod p can often be the most costly single operation in elliptic curve calculations. Traditional methods of finding inverses mod p involve applying Lagrange's Theorem or using the Extended Euclidean Algorithm (EEA). Using Lagrange's Theorem requires a very large number of modular squarings and multiplications. Because this implementation is straightforward, we first utilize this method in our calculations. (See Appendix A for a description of using Lagrange's Theorem for finding inverses mod p.) The EEA has fewer steps but typically requires a number of long divisions. Schroeppel [9] developed a method that uses only additions, subtractions, and shifts to calculate an "almost

inverse", which is then corrected to give the true inverse.

The Almost Inverse Algorithm

If p is a large prime and a is a positive integer less than p, the algorithm produces a value c and a value k (that depends on a) such that $ac = 2^k \pmod{p}$ (Hence the name, since the product is a power of 2, rather than 1). It works as follows [9]:

Input: Large integers p, a, with p an odd prime and $0 < a < p$

Output: Large integer c and integer k such that $ac = 2^k \pmod{p}$

Initialize $C = 1$, $D = 0$, $F = a$, $G = p$

Initialize integer variable $k = 0$

Loop:

While F is even, do $F \leftarrow \frac{F}{2}$, $D \leftarrow 2D$, $k \leftarrow k + 1$;

If $F = 1$, return C and k;

If $F < G$, swap F, G and swap C, D;

If $F = G \pmod{4}$, then $\{F \leftarrow F - G, C \leftarrow C - D\}$;

Else $\{F \leftarrow F + G, C \leftarrow C + D\}$;

Goto Loop;

Return $c = C$, k.

For efficiency, we have implemented the swaps as swapping pointers.

Example:

$p = 97$, $a = 5$

Operation	F	G	C	D	k
Initialize	5	97	1	0	0
Swap $(F, G), (C, D)$	97	5	0	1	0
$F = F - G, C = C - D$	92	5	-1	1	0
$F = \frac{F}{4}, D = 4D$	23	5	-1	4	2
$F = F + G, C = C + D$	28	5	3	4	2
$F = \frac{F}{4}, D = 4D$	7	5	3	16	4
$F = F + G, C = C + D$	12	5	19	16	4
$F = \frac{F}{4}, D = 4D$	3	5	19	64	6
$F < G$, Swap	5	3	64	19	6
$F = F + G, C = C + D$	8	3	83	19	6
$F = \frac{F}{8}, D = 8D$	1	3	83	152	9

So $c = 83$ is the almost inverse, solving $5 \cdot 83 = 2^9 \pmod{97}$. (Check: $27 = 27 \pmod{97}$.) The mod 4 comparison, and the respective subtraction or addition ensures that each resulting value of F will be divisible by at least 4, which allows shifting by at least two bits.

The Fixup Algorithm

Once we have solved $ac = 2^k \pmod{p}$, we would like to find $(2^k)^{-1}$ so that we can find the true inverse of a, which is $c(2^k)^{-1}$. One way to do this would be to precompute all of the possible values of $(2^k)^{-1}$ (there are about 800 possibilities of k for our choice of p). Then, we can simply multiply by the proper inverse to recover the true inverse of $a \pmod{p}$. Alternately, there is a fairly fast algorithm, the Fixup Algorithm, that allows us to recover $(2^k)^{-1} \pmod{p}$ fairly quickly, as follows [9]:

Input: k, p, and c

Output: X such that $X = c(2^k)^{-1} \pmod{p}$

Initialize $R = -p^{-1} \pmod{2^{32}}$ (This can be precomputed. For our particular p, $R = 1$.)

Initialize $X = c$

Loop:

If $k = 0$, return X;

$J = \min(k, 32)$;

$V = R \cdot X \pmod{2^J}$;

$X = X + V \cdot p$;

$X = X/2^J$;

$K = K - J$;

Goto Loop;

Schroeppel's Method for Point Multiplication

Although the Russian Peasant Method is far more efficient than simply adding G to itself many times, we can approach the problem differently using a previously unimplemented method developed by Richard Schroeppel [8]. This method vastly reduces the number of computationally expensive calculations of reciprocals as compared to affine (to about seven or eight) while taking some advantage of the smaller number of coordinates used in affine additions. To accomplish this when computing nG, we express the number n in binary, we compute all of the powers of two times G up to the bit-length of n, and we add all of the ones that are non-zero in the binary representation of n.

Computing Successive Doublings of G

Let n be a k-bit number, and we want to compute nG. Begin in projective coordinates with G having coordinates $(x, y, 1)$. First, we compute $2G$, $4G$, $8G$, ..., $2^{k-1}G$ in a way that minimizes the number of computations of reciprocals. We recall the formulas for point doubling. If $P = (x_1, y_1, z_1)$ and $2P = (x_3, y_3, z_3)$, we have:
$t = Az_1{}^2 + 3x_1{}^2$, $u = y_1z_1$, $v = ux_1y_1$, $w = t^2 - 8v$

$x_3 = 2uw$

$y_3 = t(4v - w) - 8{y_1}^2u^2$

$z_3 = 8u^3$

Notice that z_1 divides z_3. This is true even though we are continually reducing mod p (or mod the primitive polynomial of $GF(2^n)$) in the sense that $z_1^{-1} = {z_3}^{-1} \cdot 8{y_1}^3{z_1}^2$. So, if we save the coordinate values of G, $2G$, $4G$,..., $2^{k-1}G$, we can take a final inversion of the last z-coordinate and work our way back down to recover all of the earlier z-coordinates. It's then easy to change back to affine coordinates, by multiplying each of the x and y-coordinates by their respective z-coordinate (mod p). Although this requires some extra memory, we have recovered all of the powers of two multiples of G with only one reciprocal computation.

Point Additions

Now that we have expressed all of the powers of two times G, we add the terms that correspond to the ones in the binary representation of n. For example, if $n = 13$, $n = 1101_2$, so we would add $2^3G + 2^2G + 2^0G$, leaving out the 2^1G since it corresponds to a zero in the binary representation of 13. On average, there will be $\frac{\lg(n)}{2}$ one-digits, where lg is the logarithm base 2, or the number of digits in the binary representation. If we did simple affine point addition, we would still need to compute a reciprocal for every pair of points. Instead, we pair the points and defer the calculation of the inverses as follows:

Recall the formula for addition of two points, (x_1, y_1) and (x_2, y_2). $x_3 = m^2 - x_1 - x_2$ and $y_3 = m(x_1 - x_3) - y_1$, where $m = (y_2 - y_1)(x_2 - x_1)^{-1}$. To compute this, we need to know the reciprocal $(x_2 - x_1)^{-1}$. If we store the values $\{(x_2 - x_1), (x_4 - x_3)(x_2 - x_1), (x_6 - x_5)(x_4 - x_3)(x_2 - x_1), \ldots, (x_{2j} - x_{2j-1}) \cdots (x_2 - x_1)\}$, where, of course, we are reducing mod p whenever necessary, if we take the reciprocal of the final entry and multiply that by the previous entry, we will recover $(x_{2j} - x_{2j-1})^{-1}$. Multiplying our master reciprocal by $(x_{2j} - x_{2j-1})$ gives us an overall reciprocal for the previous entry, and we continue in this manner, recovering each term's reciprocal. When we're all done, we use this to compute the values of m

for the various pairs and then the affine coordinates of the sums of the pairs. If we count multiplications, we have reduced the amount of computation from computing j reciprocals (one per pair) to one reciprocal and $3(j-1)$ multiplies.

Adding the points in pairs will produce roughly $\frac{\lg(n)}{4}$ new points, which we can again pair and repeat the same trick. We can do this repeatedly until we are down to a few points, at which time we do the final additions.

Computational Advantage

Let k be the number of bits in n.
For the Russian Peasant Method, we must perform $k-1$ doublings and approximately $\frac{k}{2}$ additions. Each of these requires a reciprocal computation, so there are roughly $\frac{3k}{2}$ reciprocals involved.

In Schroeppel's method, while computing successive doublings of G, we save not only t, u, v, w, x_3, y_3, and z_3, but also $8y_1u^2$, which is an intermediate step in computing y_3. We then notice that $z_1^{-1} = z_3^{-1} \cdot 8y_1u^2$. This means that when we compute the successive doublings, deferring finding any reciprocals until the end, we end up with a final value of z, which we invert and then multiply by the previously stored value of $8y_1u^2$, which gives us the previous z's inverse. We multiply this one by the previously stored value of $8y_1u^2$ to get the z inverse for the next previous term, and continue to iterate until we do the last inverse for the point $2G$. This method requires 1 reciprocal and $(k-1)$ multiplications, compared to $(k-1)$ reciprocals by computing the successive doublings normally.

For the affine additions, notice that we are beginning with approximately $\frac{k}{2}$ points to add, which we do in consecutive blocks of pairs. The first block comprises approximately $\frac{k}{4}$ pairs. To compute the reciprocals using our trick, we have the approximate cost:

(1 reciprocal + $3(\frac{k}{4}-1)$ multiplications)+(1 reciprocal + $3(\frac{k}{8}-1)$ multiplications)+(1 reciprocal + $3(\frac{k}{16}-1)$ multiplications)+$\cdots$+(1 reciprocal + $3(\frac{k}{2^{\lg k}}-1)$ multiplications)
Summing the series for large k yields a final answer of $(\lg k - 1)$ reciprocals + $(\frac{3k}{2} - 3\lg k)$ multiplications.

So our final amount of work, combining the doublings and the affine additions, is $\lg k$ reciprocals and $(\frac{5k}{2} - 3\lg k - 1)$ multiplications, which is substantially better than the $\frac{3k}{2}$ reciprocals required by the Russian Peasant Method.

Results

Checking the Values

Since the sizes of the numbers are so large, it was important to verify that the code was producing the correct values. Although the program *Mathematica* does not have elliptic curve operations, it does have modular arithmetic and modular inverses for large numbers. *Mathematica* was used extensively in debugging and verifying the code.

For point multiplication, since three different methods were used, they provided a consistency check for each other.

Computer Specifications

Processor: AMD Athlon(tm) 64 X2 Dual-Core Processor TK-55

1.80 GHz

Memory (RAM): 2.00 GB

System type: 32-bit Operating System

Experimental Conditions

Each of the three methods for point multiplication, Affine, Projective, and Schroeppel, was combined with the two methods for reciprocal calculation, Lagrange's Theorem and the Almost Inverse Algorithm, for a total of six cases of interest. Ten trials were performed, each trial using a different, randomly-generated large number (about 256-bits) , and for each trial, all six cases operated on the same random number. During each trial, each of the six cases was run 100 times to obtain a more accurate average number for each case.

To minimize the effect of background processes, the priority of the code execution was set to "high".

Timing Results

The results for the timing tests are as follows:

(Of note is Trial 3, in which Schroeppel's method is about $\frac{1}{3}$ faster for the almost inverse algorithm than in the other trials. This is an outlier.)

#1	Time (ms)					
Trial	Affine		Projective		Schroeppel	
	Lagrange	Almost	Lagrange	Almost	Lagrange	Almost
1	972	164	35	33	53	34
2	981	165	35	33	53	34
3	966	165	35	33	53	34
4	967	164	35	34	53	34
5	967	164	35	33	53	34
Avg	971	164	35	33	53	34

#2	Time (ms)					
Trial	Affine		Projective		Schroeppel	
	Lagrange	Almost	Lagrange	Almost	Lagrange	Almost
1	944	161	34	33	50	33
2	950	161	34	33	50	33
3	946	161	34	33	50	34
4	950	161	34	33	50	34
5	947	161	34	33	50	33
Avg	947	161	34	33	50	33

#3	Time (ms)					
Trial	Affine		Projective		Schroeppel	
	Lagrange	Almost	Lagrange	Almost	Lagrange	Almost
1	1001	170	37	34	53	22
2	1000	170	36	34	55	24
3	1002	169	36	35	53	25
4	999	169	36	34	53	20
5	994	169	36	35	53	24
Avg	999	169	36	34	53	23

#4	Time (ms)					
Trial	Affine		Projective		Schroeppel	
	Lagrange	Almost	Lagrange	Almost	Lagrange	Almost
1	924	159	34	32	49	33
2	927	158	34	32	50	33
3	924	157	34	32	50	33
4	922	158	35	32	49	33
5	922	158	34	32	50	33
Avg	924	158	34	32	50	33

#5	Time (ms)					
Trial	Affine		Projective		Schroeppel	
	Lagrange	Almost	Lagrange	Almost	Lagrange	Almost
1	967	166	35	33	52	34
2	968	165	35	33	53	34
3	973	165	36	34	54	34
4	968	166	35	33	53	34
5	967	164	35	33	53	34
Avg	969	165	35	33	53	34

#6	Time (ms)					
Trial	Affine		Projective		Schroeppel	
	Lagrange	Almost	Lagrange	Almost	Lagrange	Almost
1	954	161	35	33	51	34
2	950	161	35	33	50	33
3	950	161	35	33	50	34
4	950	163	35	33	50	34
5	956	162	35	33	50	34
Avg	952	162	35	33	50	34

#7	Time (ms)					
Trial	Affine		Projective		Schroeppel	
	Lagrange	Almost	Lagrange	Almost	Lagrange	Almost
1	953	163	36	33	50	34
2	952	163	35	33	51	33
3	953	163	35	33	50	33
4	953	162	35	33	50	34
5	962	164	35	33	50	34
Avg	955	163	35	33	50	34

#8	Time (ms)					
Trial	Affine		Projective		Schroeppel	
	Lagrange	Almost	Lagrange	Almost	Lagrange	Almost
1	996	167	36	35	53	35
2	987	167	36	34	54	35
3	985	168	36	34	53	35
4	990	168	36	36	54	34
5	986	168	36	34	53	35
Avg	989	168	36	35	53	35

#9	Time (ms)					
Trial	Affine		Projective		Schroeppel	
	Lagrange	Almost	Lagrange	Almost	Lagrange	Almost
1	976	166	36	34	53	34
2	973	167	35	34	53	34
3	976	166	38	34	53	34
4	981	167	34	34	53	34
5	974	166	36	34	53	34
Avg	976	166	36	34	53	34

#10	Time (ms)					
Trial	Affine		Projective		Schroeppel	
	Lagrange	Almost	Lagrange	Almost	Lagrange	Almost
1	969	165	35	33	53	34
2	961	170	35	34	53	34
3	962	164	35	33	53	34
4	964	165	35	33	53	34
5	961	164	35	33	53	34
Avg	963	166	35	33	53	34

For the affine, the almost inverse algorithm gives a very large improvement (of about a factor of 6) over Lagrange's Theorem. This is because the affine makes heavy use of inverse calculations.

There is a very slight improvement using the almost inverse algorithm for projective, because the projective only requires the computation of one inverse.

In Schroeppel's method, there are roughly seven inverses used; hence, the almost inverse algorithm makes an improvement of about $\frac{1}{3}$.

Using the almost inverse algorithm for reciprocal calculation, projective and Schroeppel's method are both in the 33-35 ms range, and both are close to five times faster than the affine.

Conclusions

We have implemented for the first time Rich Schroeppel's new method for point multiplication on elliptic curves. This algorithm simultaneously takes advantage of the fewer multiplications and operations intrinsic to affine point addition and reduces the number of reciprocal calculations, which are the most computationally expensive operations. With approximately seven inverses, Schroeppel's method, taken in conjunction with a trick to speed up reciprocal calculations, is as fast as projective, which only requires one inverse. We estimate that with more emphasis on optimizing reciprocal calculations, Schroeppel's method could be about 20% faster than projective coordinates for point multiplication in our implementation.

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 $(\ldots -2, -1, 0, 1, 2, \ldots)$ under addition

the non-zero rational numbers under multiplication

$\mathbb{Z}_n$: the integers modulo n (the remainder after dividing by n) under addition

$\mathbb{Z}_p{}^*$: 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, and 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 $O(G)$ or $|G|$.

A commutative group G is one where $a * b = b * a$ for all a, b in G. Note that the addition operation in elliptic curves is inherently commutative, due to the underlying geometrical construction.

Fields

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), and distributive. Some well-known fields are the rational numbers, the real numbers, the complex numbers, and $\mathbb{Z}_p$ (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 with groups, the order of a finite field is the number of elements in it.

The characteristic of a field is the smallest positive number n such that $n \cdot 1 = 0$. For finite fields, this is the underlying prime number, p, on which the field is based. Since p is essentially the same as 0 for these fields, one cannot divide by it. For fields such as the real numbers or the rationals, the characteristic is defined to be zero.

Finite Fields

Finite fields are based on an underlying finite field, $\mathbb{Z}_p$, 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 $\mathbb{Z}_2$. 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 $g(x) = 1 \cdot x^7 + 0 \cdot x^6 + 0 \cdot x^5 + 1 \cdot x^4 + 0 \cdot x^3 + 1 \cdot x^2 + 1 \cdot x + 1$. We then pick an irreducible polynomial, which is the equivalent of a prime number for integers: a polynomial is irreducible if 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 $\mathbb{Z}_2$. 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 $GF(p^n)^*$.

For our particular case, the elliptic curve will be defined over the finite field $\mathbb{Z}_p$, where p is the prime defined in the NIST curve P-256.

Lagrange's Theorem for Finding Inverses

Lagrange's Theorem states that, for a prime p and non-zero integer x such that GCD(x, p)= 1, $x^{p-1} \equiv 1 \pmod{p}$. This implies that $x^{p-2} = x^{-1} \pmod{p}$. Therefore, we can calculate the inverse of $x \pmod{p}$ simply by raising it to a large power; namely, $p - 2$. Although this avoids any long divisions, there are a lot of modular squarings and multiplications, so the overall computational cost can be large.

Appendix B: Elliptic Curve Cryptography

Key Exchange

Elliptic curve Diffie-Hellman key exchange (included in NSA's Suite B) works as follows: Alice and Bob agree on a base point G on the curve. Alice selects a large random integer a and computes aG and sends the value to Bob. Bob selects a large random number b and computes bG and sends the value to Alice. The integers a and b are kept secret by Alice and Bob, respectively. Because of the difficulty of computing the discrete log, it is considered infeasible for an eavesdropper to recover a from aG or b from bG. Alice takes the received value bG and multiplies it by a to obtain abG. Bob similarly computes baG. Since $abG = baG$, Alice and Bob now share a common secret key (which could be, for example, the x-coordinate of abG).

Digital Signature Algorithm

Initial Setup

[14] To form a digital signature from an elliptic curve, we first generate a public/private key pair. Let G be a known base point on the curve, and choose a secret key, d, a random positive integer less than n, the order (number of points) of the elliptic curve. The point $Q = dG$ is the public key, and is made known.

Signature Generation

To generate a digital signature of a message m, we first compute $h(m)$, the hash of m. In the early standard, the hash function SHA-1 was the required hash function. We then compute two integers, r and s which (combined) are the signature. This is done as follows.
First, choose a message-specific one-time random number k that is less than n, and compute $kG = (x_1, y_1)$, where, again, G is our starting point on the elliptic curve. To compute r, we convert x_1 from a field element to an integer, x. To do this, we look at the field F_q in which

x_1 is contained. If q is an odd prime, then we can simply set $x_1 = x$. However, if q is of the form 2^m, then x_1 may be interpreted as bit string of length m bits, which we convert to an integer directly. Then we can set r equal to $x \pmod{n}$.

To compute s, we let $s = k^{-1}(h(m) + dr) \pmod{n}$, where $h(m)$ is the hash value of our message and d is the digital signature private key. If either r or s should happen to be zero, we begin again with a different random k.

Signature Verification

To verify a signature, we begin with the received message, m' (represented as a bit string), the received signature for our message (represented as integers r' and s'), the elliptic curve parameters, and the public key, Q.

We begin by computing the hash value $h(m')$, using the specifiied hash function.

We compute $c = s'^{-1} \pmod{n}$ and $u_1 = h(m') \cdot c \pmod{n}$, and $u_2 = r' \cdot c \pmod{n}$.

We then compute the elliptic curve point $(x_1, y_1) = u_1 \cdot G + u_2 \cdot Q$. If $u_1G + u_2Q$ is the point at infinity, the signature is rejected.

Finally, we convert the field element x_1 to an integer x, compute $v = x \pmod{n}$, and compare v with r'. If $r' = v$, the signature is verified with a high level of confidence. If $r' \neq v$, the signature is rejected, indicating that either the message was modified, the message was incorrectly signed, or the message was sent by an imposter.

The reason that the verification works is that, for the correctly transmitted values,

$(x_1, y_1) = h(m) \cdot s^{-1}G + r \cdot s^{-1}d \cdot G = s^{-1}(h(m) + r \cdot d)G,$

where we have used $Q = dG$. However,

$s^{-1} = k \cdot (h(m) + d \cdot r)^{-1},$

so $(x_1, y_1) = kG$, and r is, indeed, the x_1 value of this point.

Appendix C: Attacks on Elliptic Curve Cryptography

Elliptic curve cryptography takes advantage of the elliptic curve discrete logarithm problem: namely, given points P of order n and $Q \epsilon \langle P \rangle$ on a elliptic curve E, finding an integer l between 0 and $n-1$ such that $Q = lP$, or $l = log_P(Q)$, is very difficult. There are several possible attacks on the encryption, although they are Monte Carlo attacks (success is not guaranteed).

The Pohlig-Hellman Attack

This attack works by computing discrete logarithms in the prime order subgroups of $\langle P \rangle$, thus simplifying the problem of computing $l = \log_P Q$. First, n is factored into primes $p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$. For each $p_x^{e_x}$, $l_x = l \ (mod \ p_x^{e_x})$. Then the Chinese Remainder Theorem can be used to compute l such that $l \equiv l_i \ (mod \ p_i^{e_i})$ for all $1 \leq i \leq r$. To compute l_i, it is written base p_i as $l_i = z_0 + z_1 p_i + \cdots + z_{e_i - 1} p_i^{e_i - 1}$. Then, we define $P_0 = \frac{n}{p_i} P$ and $Q_0 = \frac{n}{p_i} Q$, and each z_r can be computed by solving $z_r = log_{P_0} Q_r$, where $Q_r = \frac{n}{p_i^{r+1}}(Q - z_0 P - z_1 p_i P - \cdots - z_{r-1} p_i^{r-1} P)$. Thus, the best strategy for guarding against the Pohlig-Hellman attack is to make n divisible by a large prime.

Pollard's Rho Attack

This attack is useful when P has a prime order n, thus rendering the Pohlig-Hellman attack useless. In Pollard's rho algorithm, two distinct pairs of integers modulo n, say (a, b) and (a', b'), are found such that $aP + bQ = a'P + b'Q$. Gathering terms and substituting $Q = lP$ gives $(a - a')P = (b - b')Q = (b - b')lP$, so $(a - a') \equiv (b - b')l \ (mod \ n)$. Thus, $l = log_P Q$ can be computed by $l = (a - a')(b - b')^{-1} \ (mod \ n)$.

To find pairs (a, b) and (a', b'), an iterating function is used.

One starts by partitioning the elements of the multiples of G into three (or more) roughly equal parts, S_1, S_2, and S_3 in a random way, for example, one could hash the x coordinate of

kG, and compute the hash's value mod 3, and assign it to one of the three parts according to that value. Starting with $Y = P$, one iterates the point Y as follows:

$$Y_{i+1} = \mathcal{F}(Y_i) = \begin{cases} Y_i + P & Y_i \in S_1 \\ 2Y_i & Y_i \in S_2 \\ Y_i + Q & Y_i \in S_3 \end{cases}$$

And one keeps track of the values a and b, where each Y is of the form $aP + bQ$.

Graphically, the successive points may be visualized as moving along a "tail" until they fall into a cycle (a circle) where they must eventually repeat. A representation is shown below, and it also explains the source of the name of the attack.

The expected number of steps $\langle X \rangle$ until a collision (two values that match) is found can

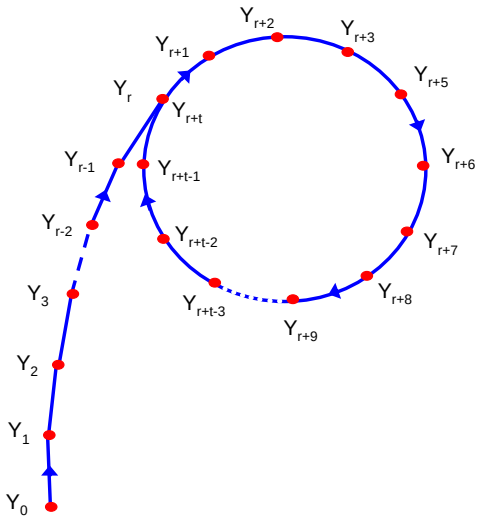

Figure 7: Pollard rho

be derived as follows. Suppose that there is a set of n possible values, and that samples are chosen at random, with replacement. The probability that there will not be a collision in k samplings is

$P[X > k] = 1 \cdot (1 - \frac{1}{n}) \cdot (1 - \frac{2}{n}) \cdots (1 - \frac{k-1}{n})$.

This is equal to the probability that there will be more than k samples required before a collision occurs. Note that there must be a collision in n (or more) samplings, both intuitively and by the formula ($P[X > n] = 0$). The probability that *exactly* k trials will be required to get a collision is then

$P[X > (k-1)] - P[X > k]$.

This allows us to write the formula for the expected number of samplings for a collision to occur,

$\langle X \rangle = \sum_{k=1}^{n} k \cdot (P[X > (k-1)] - P[X > k])$.

Expanding the sum, we have

$\langle X \rangle = (P[X > 0] - P[X > 1]) + 2(P[X > 1] - P[X > 2]) +$

$3(P[X > 2] - P[X > 3]) + \cdots + n(P[X > (n-1)] - P[X > n])$.

Since $P[X > 0] = 1 = P[X > 1]$, and $P[X > n] = 0$, terms combine to give

$\langle X \rangle = P[X > 0] + P[X > 1] + P[X > 2] + \cdots + P[X > (n-1)]$, or

$\langle X \rangle = \sum_{k=0}^{n} P[X > k]$.

At this point, we approximate $P[X > j] = 1 \cdot (1 - \frac{1}{n}) \cdot (1 - \frac{2}{n}) \cdots (1 - \frac{j-1}{n})$ by saying that $(1 - \frac{1}{n}) \approx e^{-1}$, $(1 - \frac{2}{n}) \approx e^{-2}$, ... , $(1 - \frac{j-1}{n}) \approx e^{-(j-1)}$, to get

$P[X > j] = 1 \cdot (1 - \frac{1}{n}) \cdot (1 - \frac{2}{n}) \cdots (1 - \frac{j-1}{n}) \approx e^{-1} e^{-2} \cdots e^{-(j-1)} = e^{-(1+2+\cdots+(k-1))} = e^{-\frac{k(k-1)}{2n}}$.

Now, if $k << n$, we have $e^{-\frac{k(k-1)}{2n}} \approx e^{-\frac{k^2}{2n}}(1 - O(\frac{k}{n})) \approx e^{-\frac{k^2}{2n}}$, while, if k is on the order of n (or larger), $e^{-\frac{k(k-1)}{2n}}$ is insignificantly small. In fact, we may freely extend our sum for $\langle X \rangle$ to infinity, and obtain

$\langle X \rangle \approx \sum_{k=0}^{\infty} e^{-\frac{k^2}{2n}} \approx \int_{0}^{\infty} e^{-\frac{x^2}{2n}} dx = \sqrt{\frac{\pi n}{2}}$.

In summary, the Pollard rho attack is a "square root" attack–it is proportional to the square root of the group size. This is not particularly efficient compared with attacks on factoring numbers, and it helps explain why elliptic curve cryptologic systems are considered to be secure with relatively small key sizes.

References

[1] Cohen and Frey.

Handbook of Elliptic and Hyperelliptic Curve Cryptography. Chapman & Hall / CRC (2006). Boca Raton, Forida.

[2] Hankerson, Menezes, Vanstone.

Guide to Elliptic Curve Cryptography. Springer-Verlag (2004). New York.

[3] Knuth, Donald.

The Art of Computer Programming, vol. 1, Fundamental Algorithms. Addison-Wesley (1997). Reading, Massachusetts.

[4] Knuth, Donald.

The Art of Computer Programming, vol. 2, Fundamental Algorithms. Addison-Wesley (1981). Reading, Massachusetts.

[5] Knuth, Donald.

The Art of Computer Programming, vol. 3, Sorting and Searching. Addison-Wesley (1998). Reading, Massachusetts.

[6] Menezes, Alfred.

Elliptic Curve Public Key Cryptosystems. Kluwer Academic Publishers (1993). Boston.

[7] van Oorschot, Paul C. and Wiener, Michael J.

Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology, Volume 12, Number 1.* Springer-Verlag (1999). New York.

[8] Schroeppel, Rich.

A Method For Speeding Up Point Multiplication On Elliptic Curves. Private communication.

[9] Schroeppel, Rich.

Fast Modular Reciprocals. Unpublished manuscript (1995).

[10] Teske, Edlyn.

On Random Walks for Pollard's Rho Method. *Mathematics of Computation, Volume 70, Number 234*. Electronically published (2000).

[11] Washington, Lawrence.

Elliptic Curves: Number Theory and Cryptography, 2nd ed. CRC Press (2008). New York.

[12] Wiener, Michael J.

The Full Cost of Cryptanalytic Attacks. *Journal of Cryptology, Volume 17, Number 2*. Springer-Verlag (2004). New York.

[13] Federal Information Processing Standards Publication 186-2.

Digital Signature Standard (DSS). National Institute of Standards and Technology (2000).

[14] Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).

AMERICAN NATIONAL STANDARD (ANSI) X9.62-1998.

[15] http://www.nsa.gov/ia/programs/suiteb/cryptography_index.shtml.

Source Code

```
/**********************************
    Affine, Projective, Schroeppel
    Lagrange, Almost Inverse
    Kristin Cordwell, Chen Zhao
    Last updated: 3-31-2009
**********************************/

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

const int NBIT=32;  // number of bits in a unsigned long int
const int NC=8;     // number of coefficients in a smallNum (defined below)
const int NP=256;   // number of points (for schroeppel's method)

// define cut = 2^32-1
const unsigned long long int cut = 0x00000000ffffffff;

// define structs
typedef struct {
    unsigned long int coeff[2*NC];
    int deg;
    int sign;
} bigNum;       // up to 2^512-1

typedef struct {
    unsigned long int coeff[NC];
    int deg;
    int sign;
} smallNum;     // up to 2^256-1

typedef struct {
    smallNum x, y, z;
} pcoord;

typedef struct {
    smallNum x, y;
} acoord;

// -------- global variables ---------- //
smallNum p, pb;         // p + pb = 2^256
smallNum zero, one, two, three;
bigNum big_one;
int inv_method;
unsigned long int table[NBIT+1];

// -------- function prototypes --------- //
void initialize();
int call_mult();
void call_inv();
void build_table();
void read(smallNum*, smallNum*, smallNum*);
smallNum multiply(smallNum*, smallNum*);
bigNum multiply_noreduce(smallNum*, smallNum*);
smallNum reduce(bigNum*);
void reduce_mod_p(smallNum*);
smallNum add_and_reduce(smallNum*, smallNum*);
```

```
smallNum sub_and_reduce(smallNum*, smallNum*);
bigNum add_big(bigNum*, bigNum*);
bigNum sub_big(bigNum*, bigNum*);
smallNum lagrange(smallNum*);
smallNum m_add(smallNum*, smallNum*, smallNum*, smallNum*);
smallNum x_add(smallNum*, smallNum*, smallNum*);
smallNum y_add(smallNum*, smallNum*, smallNum*, smallNum*);
smallNum m_double(smallNum*, smallNum*);
smallNum x_double(smallNum*, smallNum*);
smallNum y_double(smallNum*, smallNum*, smallNum*, smallNum*);
void point_add(smallNum*, smallNum*, smallNum*, smallNum*);
void point_double(smallNum*, smallNum*);
void affine_mult(smallNum*, smallNum*, smallNum*);
void projective_add(smallNum*, smallNum*, smallNum*, smallNum*, smallNum*, \
                    smallNum*, smallNum*, smallNum*, smallNum*);
void projective_double(smallNum*, smallNum*, smallNum*, smallNum*, \
                    smallNum*, smallNum*, smallNum*);
void mixed_add(smallNum*, smallNum*, smallNum*, smallNum*, smallNum*, \
                    smallNum*, smallNum*, smallNum*);
void projective_mult(smallNum*, smallNum*, smallNum*);
bool compare(smallNum*, smallNum*);
bool compare(bigNum*, bigNum*);
bool isOne(bigNum*);
int firstbig(smallNum*, smallNum*);
int firstbig(bigNum*, bigNum*);
void convert(smallNum*, smallNum*, smallNum*, smallNum*, smallNum*);
int find_degree(smallNum*);
int find_degree(bigNum*);
void outputSML_nospace(smallNum*);
void output_for_input(smallNum*);
void outputBIG_nospace(bigNum*);
smallNum affine_addx(smallNum*, smallNum*, smallNum*);
smallNum affine_addy(smallNum*, smallNum*, smallNum*, smallNum*);
void schroeppel(smallNum*, smallNum*, smallNum*);
smallNum almost(smallNum*);


int main()
{
    smallNum x1, y1, k;      // x, y coord of g and random factor
    clock_t time1, time2;    // start and stop times
    int n, i;

    initialize();            // initialize p, pb, and others
    build_table();
    read(&x1, &y1, &k);

    n = call_mult();     // point multiplication method
    call_inv();          // smallNum inversion method

    time1 = clock();
    cout << "Working . . ." << endl;

//  SWITCHES
    if(n == 1) for(i=1; i<=100; i++) affine_mult(&x1, &y1, &k);
    else if (n == 2) for(i=1; i<=100; i++) projective_mult(&x1, &y1, &k);
    else if (n == 3) for(i=1; i<=100; i++) schroeppel(&x1, &y1, &k);

    time2 = clock();
    cout << "CPU time : " << time2-time1 << " milliseconds" << endl;

    outputSML_nospace(&x1);      // write answer to file to  //
    outputSML_nospace(&y1);      // check using Mathematica  //
```

```
    return 0;   // end
}


int call_mult()
{
    int n;
    cout << "Point Mult Method:" << endl;
    cout << "1: Affine" << endl;
    cout << "2: Projective" << endl;
    cout << "3: Schroeppel" << endl;
    cin >> n;

    if(n!=1 && n!=2 && n!=3) {
        cout << "fail" << endl;
        exit(0);
    }
    cout << endl;
    return n;
}


void call_inv()
{
    cout << "1: Lagrange" << endl;
    cout << "2: Almost Inverse" << endl;
    cin >> inv_method;

    if(inv_method!=1 && inv_method!=2) {
        cout << "fail" << endl;
        exit(0);
    }
    cout << endl;
}


// initialization of smallNum p, pb, zero, one, three
void initialize()
{
    smallNum f;

    for(int i=0; i<NC; i++) {
        f.coeff[i]=0xffffffff;
        if(i<=2) p.coeff[i]=0xffffffff;
        else if(i<=5) p.coeff[i]=0;
        else if(i==6) p.coeff[i]=1;
        else p.coeff[i]=0xffffffff;
        pb.coeff[i]=f.coeff[i]-p.coeff[i];

        zero.coeff[i]=0;
        one.coeff[i]=0;
        two.coeff[i]=0;
        three.coeff[i]=0;
    }
    pb.coeff[0] = 1;

    one.coeff[0] = 1;
    two.coeff[0] = 2;
    three.coeff[0] = 3;

    zero.deg = 0;
    one.deg = 0;
    two.deg = 1;
    three.deg = 1;
```

```
    for(int i=1; i<15; i++)
        big_one.coeff[i] = 0;
    big_one.coeff[0] = 1;
}


void build_table()
{
    for(int i=0; i<NBIT; i++)
        table[i] = (1 << i) - 1;
    table[NBIT] = 0xffffffff;
}

void read(smallNum *a, smallNum *b, smallNum *c)
{
    int hi_byt, hi_bit;
    short bit;
    fstream fin;
    smallNum *temp;
    char aline[256], ch;

// initialize a, b, and c to zero
    a->deg = 0;
    b->deg = 0;
    c->deg = 0;
    for(int i=0; i<NC; i++) {
        a->coeff[i] = 0;
        b->coeff[i] = 0;
        c->coeff[i] = 0;
    }

// open input data file
    fin.open("datafiles/gmult_in.dat");
    for(int i=0; i<3; i++) {
        if(i==0) temp = a;
        else if(i==1) temp = b;
        else if(i==2) temp = c;

        ch = fin.peek();
        while(ch < '0' || ch > '9') {
            fin.getline(aline, 256);
            ch = fin.peek();
        }

        fin >> temp->deg;

        hi_byt=temp->deg/NBIT;
        hi_bit=temp->deg%NBIT;

        temp->coeff[hi_byt]=0x00000000; // initialize partial long int

        // partial long int
        for(int i=hi_bit; i>=0; i--) {
            fin >> bit;
            if(bit==1) temp->coeff[hi_byt]^=(0x00000001<<i);
        }

        // full long ints
        for(int i=hi_byt-1; i>=0; i--) {
            temp->coeff[i]=0x00000000; // initialize full long ints
            for(int j=NBIT-1; j>=0; j--) {
                fin >> bit;
                if(bit==1) temp->coeff[i]^=(0x00000001<<j);
```

```
            }
        }
        fin.ignore();
    }
}


smallNum multiply(smallNum *a, smallNum *b)
{
    bigNum c;
    smallNum c_final;
    unsigned long long int at, bt, ct;
    unsigned long long int temp;
    int carry[2*NC+1];

    // initialization
    for(int i=0; i<2*NC; i++) {
        c.coeff[i]=0;
        carry[i]=0;
    }

    for(int i=0; i<NC; i++) {
        for(int j=0; j<NC; j++) {
            at=a->coeff[i];
            bt=b->coeff[j];
            ct=at*bt;

            /*calculate c.coeff[i+j]*/
            temp=ct&cut;
            temp=temp+c.coeff[i+j]+carry[i+j];
            carry[i+j]=0;
            if(temp>cut) {
                carry[i+j+1] += 1;
                temp -= (cut+1);
            }
            c.coeff[i+j] = temp;

            /*calculate c.coeff[i+j+1]*/
            temp=ct>>32;
            temp=temp+c.coeff[i+j+1]+carry[i+j+1];
            carry[i+j+1]=0;
            if(temp>cut) {
                carry[i+j+2] += 1;
                temp -= (cut+1);
            }
            c.coeff[i+j+1] = temp;
        }
    }

    c_final = reduce(&c);
    c_final.deg=find_degree(&c_final);
    return c_final;
}


smallNum add_and_reduce(smallNum *a, smallNum *b)
{
    smallNum c;
    unsigned long long int at, bt, temp;
    int carry[9];

    c.sign = a->sign;

    // initialization of carry and c
```

```
    for(int i=0; i<8; i++) {
        c.coeff[i]=0;
        carry[i]=0;
    }
    carry[8]=0;

    for(int i=0; i<8; i++) {
        at = a->coeff[i];
        bt = b->coeff[i];
        temp = at + bt + carry[i];
        if(temp>cut) {          // temp should always be less than cut
            temp = temp - cut - 1;
            carry[i+1] = 1;     // carry over to the next coefficient
        }
        c.coeff[i] = temp;
    }

    // check if sum (c) is larger than smallNum \
    // can hold, if yes, reduce by p.
    // "reduce by p" is equavilent to "add pb" \
    // and "drop the carryover" (already dropped).
    if(carry[8] == 1) {
        for(int i=0; i<=8; i++) carry[i] = 0;  // re-initialization
        for(int i=0; i<8; i++) {
            at = c.coeff[i];
            bt = pb.coeff[i];
            temp = at + bt + carry[i];
            if(temp>cut) {
                temp = temp - cut - 1;
                carry[i+1] = 1;     // carry over to next coefficient
            }
            c.coeff[i] = temp;
        }
    }

    // it is possible to need to reduce c by p one last time
    if(carry[8] == 1) {
        for(int i=0; i<=8; i++) carry[i] = 0;  // re-initialization
        for(int i=0; i<8; i++) {
            at = c.coeff[i];
            bt = pb.coeff[i];
            temp = at + bt + carry[i];
            if(temp>cut) {
                temp = temp - cut - 1;
                carry[i+1] = 1;     // carry over to next coefficient
            }
            c.coeff[i] = temp;  // temp > 2^256
        }
    }

    return c;
}


smallNum sub_and_reduce(smallNum *a, smallNum *b)
{
    smallNum c;
    unsigned long long int temp;
    int borrow[9];

    c.sign = a->sign;

    // initialization of borrow and c
    for(int i=0; i<8; i++) {
```

```
        c.coeff[i]=0;
        borrow[i]=0;
    }
    borrow[8]=0;

    for(int i=0; i<8; i++) {
        // to avoid 0-1 = 2^32-1 w/o borrowing
        if(a->coeff[i]==0 && borrow[i]==1) {
            a->coeff[i] = cut + 1;
            borrow[i+1] = 1;
        }
        if(a->coeff[i] - borrow[i] >= b->coeff[i]) {
            c.coeff[i] = a->coeff[i] - borrow[i] - b->coeff[i];
        } else {
            temp = a->coeff[i] + cut + 1;
            c.coeff[i] = temp - b->coeff[i] - borrow[i];
            borrow[i+1] = 1;
        }
    }

    // a < b, borrowed 2^256, need to add p,
    // which is to subtract pb from c.
    while(borrow[8]==1) {
        for(int i=0; i<=8; i++) borrow[i]=0;
        for(int i=0; i<8; i++) {
            // to avoid 0-1 = 2^32-1 w/o borrowing
            if(c.coeff[i]==0 && borrow[i]==1) {
                c.coeff[i] = cut + 1;
                borrow[i+1] = 1;
            }
            if(c.coeff[i] - borrow[i] >= pb.coeff[i]) {
                c.coeff[i] = c.coeff[i] - borrow[i] - pb.coeff[i];
            } else {
                temp = c.coeff[i] + cut + 1;
                c.coeff[i] = temp - pb.coeff[i] - borrow[i];
                borrow[i+1] = 1;
            }
        }
    }
    return c;
}


// find the degree of smallNum with NC coefficients
int find_degree(smallNum *a)
{
    int degree;
    for(int i=NC-1; i>=0; i--) {
        if(a->coeff[i]>0) {
            for(int j=31; j>=0; j--) {
                if(a->coeff[i]>>j == 1) {
                    degree = 32*i+j;
                    return degree;
                }
            }
        }
    }
    return 0;
}


// find the degree of bigNum with 2*NC coefficients
int find_degree(bigNum *a)
{
```

```
    int degree;
    for(int i=2*NC-1; i>=0; i--) {
        if(a->coeff[i]>0) {
            for(int j=31; j>=0; j--) {
                if(a->coeff[i]>>j == 1) {
                    degree = 32*i+j;
                    return degree;
                }
            }

        }
    }
    return 0;
}


bool compare(smallNum *a, smallNum *b)
{
    for(int i=7; i>=0; i--) {
        if(a->coeff[i] != b->coeff[i]) return false;
    }

    return true;
}


void reduce_mod_p(smallNum *a)
{
    int abigger=0;

    for(int i=7; i>=0; i--) {
        if(a->coeff[i] < p.coeff[i]) {
            abigger = -1;
            break;
        } else if(a->coeff[i] > p.coeff[i]) {
            abigger = 1;
            break;
        }
    }
    if(abigger==0) {
        for(int i=0; i<8; i++) a->coeff[i]=0;
    } else if(abigger==1) {
        *a=sub_and_reduce(a, &p);
    }
}


void outputSML_nospace(smallNum *B)
{
    unsigned long int num;
    int hi_byt, hi_bit;
    int i, j;
    static ofstream fout;

    if(!fout.is_open()) fout.open("datafiles/smlout_nospace.dat");

    B->deg = find_degree(B);
    fout << B->deg << endl;
    hi_byt=B->deg/32;
    hi_bit=B->deg%32;

    // partial long long int
    for(i=hi_bit; i>=0; i--) {
        num = (B->coeff[hi_byt]>>i)&(0x00000001);
```

```
        fout << num;
    }

    // full long ints
    for(i=hi_byt-1; i>=0; i--) {
        for(j=31; j>=0; j--) {
            num = (B->coeff[i]>>j)&(0x00000001);
            fout << num;
        }
    }
    fout << endl;
}


void outputBIG_nospace(bigNum *B)
{
    unsigned long int num;
    int hi_byt, hi_bit;
    int i, j;
    static ofstream fout;

    if(!fout.is_open()) fout.open("datafiles/bigout_nospace.dat");

    B->deg = find_degree(B);
    fout << B->deg << endl;

    hi_byt=B->deg/32;
    hi_bit=B->deg%32;

    // partial long long int
    for(i=hi_bit; i>=0; i--) {
        num = (B->coeff[hi_byt]>>i)&(0x00000001);
        fout << num;
    }

    // full long ints
    for(i=hi_byt-1; i>=0; i--) {
        for(j=31; j>=0; j--) {
            num = (B->coeff[i]>>j)&(0x00000001);
            fout << num;
        }
    }
    fout << endl;
}


smallNum lagrange(smallNum *a)
{
    smallNum order, b;

    // initialize b = 1 and order = p - 2;
    for(int i=NC-1; i>0; i--) {
        b.coeff[i] = 0;
        order.coeff[i]=p.coeff[i];
    }
    b.coeff[0] = 1;          // b = 1
    order.coeff[0] = p.coeff[0] - 2;    // order = p - 2

    // calculate inverse of a mod p using
    // lagrange's theorem with russian peasant technique

    for(int i=NC-1; i>=0; i--) {
        for(int j=NBIT-1; j>=0; j--) {
            if((((order.coeff[i]) >> j) & 1) == 1) {
```

```
                // if binary digit is 1, square then multiply by a
                b = multiply(&b, &b);   // square
                b = multiply(a, &b);    // multiply
            } else {        // if binary is 0, just square
                b = multiply(&b, &b);   // square
            }
        }
    }
    return b;
}


//********************** Affine Algorithm ********************** //
smallNum m_add(smallNum *px, smallNum *py, smallNum *qx, smallNum *qy)
{
    smallNum dx, dy, dx_inv, m;

    // find dx (denominator of m)
    dx = sub_and_reduce(qx, px);    // dx = qx - px
    if(inv_method == 1) dx_inv = lagrange(&dx);     // this is dx^(-1)
    else if(inv_method == 2) dx_inv = almost(&dx);

    // find dy (numerator of m)
    dy = sub_and_reduce(qy, py);    // dy = qy - py

    // find m = dy * dx^(-1)
    m = multiply(&dy, &dx_inv);
    return m;
}


// rx = m^2 - px - qx
smallNum x_add(smallNum *m, smallNum *px, smallNum *qx)
{
    smallNum rx;

    rx = multiply(m, m);
    rx = sub_and_reduce(&rx, px);
    rx = sub_and_reduce(&rx, qx);

    return rx;
}


// ry = m*(qx-rx) - qy
smallNum y_add(smallNum *m, smallNum *rx, smallNum *qx, smallNum *qy)
{
    smallNum ry;

    ry = sub_and_reduce(qx, rx);
    ry = multiply(&ry, m);
    ry = sub_and_reduce(&ry, qy);
    return ry;
}


void point_add(smallNum *px, smallNum *py, smallNum *qx, smallNum *qy)
{
    // add p and q
    smallNum m, rx, ry;
    m = m_add(px, py, qx, qy);  // find m (slope mod p)
    rx = x_add(&m, px, qx);     // rx = m^2 - px - qx
    ry = y_add(&m, &rx, qx, qy);    // ry = m(qx-rx) - qy
    // answers are rx and ry, which are to be copied back to px and py
```

```
    *px = rx;
    *py = ry;
}


smallNum m_double(smallNum *gx, smallNum *gy)
{
    smallNum dx, dy, dx_inv, m;

    // find dx (denominator of m)
    dx = add_and_reduce(gy, gy);
    if(inv_method == 1) dx_inv = lagrange(&dx);     // find dx^(-1)
    else if(inv_method == 2) dx_inv = almost(&dx);

    // find dy (numerator of m)
    dy = multiply(gx, gx);      // gx^2
    dy = multiply(&dy, &three);
    dy = sub_and_reduce(&dy, &three);   // dy = 3*gx^2 - 3

    // find m = dy * dx^(-1)
    m = multiply(&dy, &dx_inv);
    return m;
}


smallNum x_double(smallNum *m, smallNum *px)
{
    smallNum rx;

    rx = multiply(m, m);    // get m^2 mod p
    rx = sub_and_reduce(&rx, px);  // subtract 2*px
    rx = sub_and_reduce(&rx, px);

    return rx;
}


smallNum y_double(smallNum *m, smallNum *px, smallNum *py, smallNum *rx)
{
    smallNum ry;

    ry = sub_and_reduce(px, rx);
    ry = multiply(m, &ry);
    ry = sub_and_reduce(&ry, py);

    return ry;
}


void point_double(smallNum *gx, smallNum *gy)
{
    smallNum m, hx, hy;
    m = m_double(gx, gy);       // find m (slope mod p)
    hx = x_double(&m, gx);      // hx = m^2 - 2*gx
    hy = y_double(&m, gx, gy, &hx); // hy = m(gx-hx) - gy
    // answers are hx and hy, which are to be copied back to gx and gy
    *gx = hx;
    *gy = hy;
}


// russian peasant method for multiplication
void affine_mult(smallNum *gx, smallNum *gy, smallNum *factor)
{
```

```
    // need declaration of temporary identifiers
    smallNum xtemp, ytemp;
    int s, highest;

    xtemp=*gx;
    ytemp=*gy;

    highest = (factor->deg)/NBIT;   // the highest array with coefficients
    // russian peasant starts with first 1 in k
    for(int i=highest; i>=0; i--) {
        if(i==highest) s=factor->deg%NBIT;
        else s=NBIT;

        for(int j=s-1; j>=0; j--) {
            if((((factor->coeff[i]) >> j) & 1) == 1) {
                // if digit is 1, double then add g
                point_double(&xtemp, &ytemp);
                point_add(&xtemp, &ytemp, gx, gy);
            } else {    // if digit is 0, just double
                point_double(&xtemp, &ytemp);
            }
        }
    }

    // copy answers back to gx and gy
    *gx = xtemp;
    *gy = ytemp;
}


// ****************** Projective Algorithm ******************** //
void projective_add(smallNum *x1, smallNum *y1, smallNum *z1,\
                    smallNum *x2, smallNum *y2, smallNum *z2,\
                    smallNum *x3, smallNum *y3, smallNum *z3)
// (x1, y1, z1) + (x2, y2, z2) = (x3, y3, z3)
{
    // declaration of variables/identifiers
    smallNum u1, u2, v1, v2;
    smallNum u, v, w, a;
    smallNum t1, t2, t3;
    smallNum vsquared, vcubed, vsquaredv2;

    u1 = multiply(y2, z1);  // u1 = y2*z1
    u2 = multiply(y1, z2);  // u2 = y1*z2
    v1 = multiply(x2, z1);  // v1 = x2*z1
    v2 = multiply(x1, z2);  // v2 = x1*z2

    u = sub_and_reduce(&u1, &u2);   // u = u1 - u2
    v = sub_and_reduce(&v1, &v2);   // v = v1 - v2

    // use memory to store values and minimize # of multiplications
    vsquared = multiply(&v, &v);            // find v^2
    vcubed = multiply(&vsquared, &v);       // find v^3
    vsquaredv2 = multiply(&vsquared, &v2);  // find v^2*v2

    w = multiply(z1, z2);

    // find a
    t1 = multiply(&u, &u);  // t1 = u^2*w
    t1 = multiply(&t1, &w);
    t2 = vcubed;            // t2 = v^3
    t3 = add_and_reduce(&vsquaredv2, &vsquaredv2);  // t3 = 2*v^2*v2

    a = sub_and_reduce(&t1, &t2);
```

```
    a = sub_and_reduce(&a, &t3);

    // needed for y3
    t1 = sub_and_reduce(&vsquaredv2, &a);
    t1 = multiply(&t1, &u);          // t1 = u*(v^2*v2 - a)
    t2 = multiply(&vcubed, &u2);     // t2 = v^3*u2

    // answers:
    *x3 = multiply(&v, &a);          // x3 = v*a
    *y3 = sub_and_reduce(&t1, &t2);
    *z3 = multiply(&vcubed, &w);     // z3 = v^3*w

    x3->deg=find_degree(x3);
    y3->deg=find_degree(y3);
    z3->deg=find_degree(z3);
    // 15 multiplications total
}


void projective_double(smallNum *x1, smallNum *y1, smallNum *z1,\
                       smallNum *x2, smallNum *y2, smallNum *z2,\
                       smallNum *zfactor)
// double (x1, y1, z1) and get (x2, y2, z2)
{
    // declarations;
    smallNum w, s, b, h;
    smallNum b2, b4, b8;
    smallNum t1, t2;
    smallNum ssquared, scubed;
//  if(y1 == 0) point at infinity

    t1 = multiply(x1, x1);
    t2 = multiply(z1, z1);
    t1 = sub_and_reduce(&t1, &t2);
    t2 = add_and_reduce(&t1, &t1);
    w = add_and_reduce(&t2, &t1);

    s = multiply(y1, z1);
    ssquared = multiply(&s, &s);
    scubed = multiply(&s, &ssquared);

    b = multiply(x1, y1);
    b = multiply(&b, &s);

    b2 = add_and_reduce(&b, &b);
    b4 = add_and_reduce(&b2, &b2);
    b8 = add_and_reduce(&b4, &b4);

    h = multiply(&w, &w);
    h = sub_and_reduce(&h, &b8);

    *x2 = multiply(&h, &s);
    *x2 = add_and_reduce(x2, x2);

    t1 = sub_and_reduce(&b4, &h);
    t1 = multiply(&t1, &w);


//  t2 = y1 * (8 y1 ssquared) or y1 * zfactor
    t2 = multiply(y1, &ssquared);
    t2 = add_and_reduce(&t2, &t2);
    t2 = add_and_reduce(&t2, &t2);
    t2 = add_and_reduce(&t2, &t2);
    *zfactor = t2;
```

```
    t2 = multiply(y1, &t2);

    *y2 = sub_and_reduce(&t1, &t2);

    *z2 = add_and_reduce(&scubed, &scubed);
    *z2 = add_and_reduce(z2, z2);
    *z2 = add_and_reduce(z2, z2);

    // 12 multiplications total
}


void mixed_add(smallNum *x1, smallNum *y1, smallNum *z1,\
               smallNum *x2, smallNum *y2,\
               smallNum *x3, smallNum *y3, smallNum *z3)
{
    // declaration of variables/identifiers
    smallNum u1, u2, v1, v2;
    smallNum u, v, w, a;
    smallNum t1, t2, t3;
    smallNum vsquared, vcubed, vsquaredv2;

    u1 = multiply(y2, z1);  // u1 = y2*z1
    u2 = *y1;               // u2 = y1*z2 = y1
    v1 = multiply(x2, z1);  // v1 = x2*z1
    v2 = *x1;               // v2 = x1*z2 = x1

    u = sub_and_reduce(&u1, &u2);   // u = u1 - u2
    v = sub_and_reduce(&v1, &v2);   // v = v1 - v2

    // use memory to store values and minimize # of multiplications
    vsquared = multiply(&v, &v);            // find v^2
    vcubed = multiply(&vsquared, &v);       // find v^3
    vsquaredv2 = multiply(&vsquared, &v2);  // find v^2*v2

    w = *z1;    // w = z1*z2 = z1

    // find a
    t1 = multiply(&u, &u);      // t1 = u^2*w
    t1 = multiply(&t1, &w);
    t2 = vcubed;        // t2 = v^3
    t3 = multiply(&vsquared, &v2);
    t3 = add_and_reduce(&t3, &t3);  // t3 = 2*v^2*v2

    a = sub_and_reduce(&t1, &t2);
    a = sub_and_reduce(&a, &t3);

    // needed for y3
    t1 = sub_and_reduce(&vsquaredv2, &a);
    t1 = multiply(&t1, &u);     // t1 = u*(v^2*v2 - a)
    t2 = multiply(&vcubed, &u2);    // t2 =

    // answers:
    *x3 = multiply(&v, &a);         // x3 = v*a
    *y3 = sub_and_reduce(&t1, &t2); // y3 = u*(v^2*v2 - a) - v^3*u2
    *z3 = multiply(&vcubed, &w);    // z3 = v^3*w

    x3->deg=find_degree(x3);
    y3->deg=find_degree(y3);
    z3->deg=find_degree(z3);
    // 12 multiplications total
}
```

```
void projective_mult(smallNum *x, smallNum *y, smallNum *k)
{
    smallNum z, xtemp, ytemp, ztemp, zfactor;
    int s, bit, highest;

    z = one;

    xtemp=*x;
    ytemp=*y;
    ztemp=z;

    highest = (k->deg)/NBIT;    // the highest array with coefficients

    for(int i=highest; i>=0; i--) {
        // russian peasant starts with first 1 in k
        if(i==highest) s=k->deg%NBIT;
        else s=NBIT;

        for(int j=s-1; j>=0; j--) {
            bit = (((k->coeff[i]) >> j) & 1);
            // always double
            projective_double(&xtemp, &ytemp, &ztemp, &xtemp, &ytemp, &ztemp, \
                              &zfactor);

            if(bit==1)  // add if bit is 1
                projective_add(&xtemp, &ytemp, &ztemp, x, y, &z, \
                               &xtemp, &ytemp, &ztemp);
        }
    }

    // copy answers back to gx and gy
    *x = xtemp;
    *y = ytemp;
    z = ztemp;

    convert(x, y, &z, x, y); // convert back to affine using lagrange
}


// convert from projective to affine
void convert(smallNum *px, smallNum *py, smallNum *pz, smallNum *ax, smallNum *ay)
{
    smallNum inv;

    if(inv_method == 1) inv = lagrange(pz);
    else if(inv_method == 2) inv = almost(pz);

    *ax = multiply(px, &inv);
    *ay = multiply(py, &inv);
}


void output_for_input(smallNum *B)
{
    unsigned long int num;
    int hi_byt, hi_bit;
    int i, j;
    static ofstream fout;

    if(!fout.is_open()) fout.open("datafiles/smlout_input.dat");

    B->deg = find_degree(B);
    fout << B->deg << endl;
```

```
    hi_byt=B->deg/32;
    hi_bit=B->deg%32;

    // partial long long int
    for(i=hi_bit; i>=0; i--) {
        num = (B->coeff[hi_byt]>>i)&(0x00000001);
        fout << num << " ";
    }
    fout << endl;

    // full long ints
    for(i=hi_byt-1; i>=0; i--) {
        for(j=31; j>=0; j--) {
            num = (B->coeff[i]>>j)&(0x00000001);
            fout << num << " ";
        }
        fout << endl;
    }
}


smallNum reduce(bigNum *A)
{
    int i;
    smallNum T, S1, S2, S3, S4, D1, D2, D3, D4;
    smallNum B;

    for(i=0; i<NC; i++) {
        T.coeff[i]=A->coeff[i];

        if(i<=2) S1.coeff[i]=0;
        else S1.coeff[i]=A->coeff[i+8];

        if(i<=6 && i>=3) S2.coeff[i]=A->coeff[i+9];
        else S2.coeff[i]=0;

        if(i<=5 && i>=3) S3.coeff[i]=0;
        else S3.coeff[i]=A->coeff[i+8];

        if(i<=2) S4.coeff[i]=A->coeff[i+9];
        else if(i<=5) S4.coeff[i]=A->coeff[i+10];
        else if(i==6) S4.coeff[i]=A->coeff[13];
        else S4.coeff[i]=A->coeff[8];

        if(i<=2) D1.coeff[i]=A->coeff[i+11];
        else if(i<=5) D1.coeff[i]=0;
        else if(i==6) D1.coeff[i]=A->coeff[8];
        else D1.coeff[i]=A->coeff[10];

        if(i<=3) D2.coeff[i]=A->coeff[i+12];
        else if(i<=5) D2.coeff[i]=0;
        else if(i==6) D2.coeff[i]=A->coeff[9];
        else D2.coeff[i]=A->coeff[11];

        if(i<=2) D3.coeff[i]=A->coeff[i+13];
        else if(i<=5) D3.coeff[i]=A->coeff[i+5];
        else if(i==6) D3.coeff[i]=0;
        else D3.coeff[i]=A->coeff[12];

        if(i<=1) D4.coeff[i]=A->coeff[i+14];
        else if(i==2) D4.coeff[i]=0;
        else if(i<=5) D4.coeff[i]=A->coeff[i+6];
        else if(i==6) D4.coeff[i]=0;
        else D4.coeff[i]=A->coeff[13];
```

```
    }

    B=add_and_reduce(&T, &S1);
    B=add_and_reduce(&B, &S1);
    B=add_and_reduce(&B, &S2);
    B=add_and_reduce(&B, &S2);
    B=add_and_reduce(&B, &S3);
    B=add_and_reduce(&B, &S4);

    B=sub_and_reduce(&B, &D1);
    B=sub_and_reduce(&B, &D2);
    B=sub_and_reduce(&B, &D3);
    B=sub_and_reduce(&B, &D4);

    reduce_mod_p(&B);

    B.deg=find_degree(&B);

    return B;
}


// ************** new function for affine addition **************
smallNum affine_addx(smallNum *qx, smallNum *px, smallNum *m)
{
    smallNum msquared, rx;

    msquared = multiply(m, m);
    rx = sub_and_reduce(&msquared, qx);
    rx = sub_and_reduce(&rx, px);

    return rx;
}


// ry = m(qx-rx) - qy
smallNum affine_addy(smallNum *rx, smallNum *qx, smallNum *qy, smallNum *m)
{
    smallNum ry;

    ry = sub_and_reduce(qx, rx);
    ry = multiply(&ry, m);
    ry = sub_and_reduce(&ry, qy);

    return ry;
}


void schroeppel(smallNum *x, smallNum *y, smallNum *k)
{
    pcoord power[NP];
    int i, j, nd, index;
    int count = 0;      // for counting number of 1's in gfactor
    smallNum z_inv[NP], zfactor[NP];
    acoord needed[NP];  // very unlikely for needed to use all NP slots
    smallNum d[NP/2];   // delta x's
    smallNum e[NP/2];   // delta y's
    smallNum m[NP/2];   // m is the slope, which equals (delta y) / (delta x)
    smallNum *d_inv = NULL;
    smallNum *d_group = NULL;
    smallNum *d_group2 = NULL;
    smallNum D_inv, temp;

    power[0].x = *x;
```

```
    power[0].y = *y;
    power[0].z = one;

    //  --------------------------- Part I ----------------------------
    // generate G's: power[0] = G, power[1] = 2G, ..., power[i] = (2^i)G
    for(i=0; i<NP-1; i++) {      // NP = 256
        projective_double(&power[i].x, &power[i].y, &power[i].z, \
            &power[i+1].x, &power[i+1].y, &power[i+1].z, &zfactor[i]);
    }

    if(inv_method == 1) z_inv[NP-1] = lagrange(&power[NP-1].z);
    else if(inv_method == 2) z_inv[NP-1] = almost(&power[NP-1].z);

    for(i=NP-2; i>=0; i--) { // first i = 254, last i=0
        z_inv[i] = multiply(&z_inv[i+1], &zfactor[i]);
    }

    // convert all needed coordinates to affine by multiplying
    for(i=0; i<NC; i++) {    // NC = 8
        for(j=0; j<NBIT; j++) { // NBIT = 32
            if((((k->coeff[i]) >> j) & 1) == 1) {
                index = NBIT*i+j;
                needed[count].x = multiply(&z_inv[index], &power[index].x);
                needed[count].y = multiply(&z_inv[index], &power[index].y);
                count += 1;
            }
        }
    }

//  ------------------------ Part II -------------------------

    while(count!=1) {
        for(i=0; i<count; i+=2) {
            d[i/2]=sub_and_reduce(&needed[i+1].x, &needed[i].x);
            e[i/2]=sub_and_reduce(&needed[i+1].y, &needed[i].y);
        }   // generate d's (delta x's) and e's (delta y's)

        d_group = new smallNum[count];// multiplied groups of d

        nd = count/2;   // number of d's
        d_group[0] = d[0];

        // generate d_group
        for(i=1; i<nd; i++) {
            d_group[i] = multiply(&d_group[i-1], &d[i]);
        }   //  d_group[0] = d[0], d_group[1]=d[0]d[1], ...,


        smallNum D = d_group[nd-1];
        if(inv_method == 1) D_inv = lagrange(&D);
        else if(inv_method == 2) D_inv = almost(&D);

//  ---------------------------- Part III ----------------------------
        d_group2 = new smallNum[nd];

        d_group2[nd-1] = D_inv;


        for(i=nd-2; i>=0; i--) {    // generate d_group2;
            d_group2[i] = multiply(&d_group2[i+1], &d[i+1]);
        }   // d_group2[nd-1] = D_inv,  d_group2[nd-2] = D_inv*d[nd-1]
            // d_group2[1] = D_inv*d[nd-1]*d[nd-2]*...*d[1]

        d_inv = new smallNum[nd];
```

```
        //  d_group[0] = d[0], d_group[1]=d[0]d[1], ...,
        //  d_group[nd-1] = d[0]d[1]...d[nd-1]

        for(i=nd-1; i>=1; i--) {    // find inverses of all d's
            d_inv[i] = multiply(&d_group2[i], &d_group[i-1]);
        }
        d_inv[0] = d_group2[0];

        for(i=0; i<nd; i++) {   //
            m[i] = multiply(&e[i], &d_inv[i]);
        }

        for(i=0; i<=count-2; i+=2) {
            temp = affine_addx(&needed[i].x, &needed[i+1].x, &m[i/2]);
            needed[i/2].y = affine_addy(&temp, &needed[i].x, &needed[i].y, &m[i/2]);
            needed[i/2].x = temp;
        }

        if(count%2 == 1) {
            needed[count/2].x = needed[count-1].x;
            needed[count/2].y = needed[count-1].y;
            count = count/2 + 1;
            nd = nd/2 + 1;
        } else {
            count = count/2;
            nd = nd/2;
        }

        delete d_group;
        delete d_inv;
    }

    *x = needed[0].x;
    *y = needed[0].y;
}


smallNum almost(smallNum *a)
{
    smallNum v, inv, c, d, stemp;
    bigNum f, g, btemp, x, ans, ans2;
    unsigned long int cftemp;   // for coeff

    int i, j, k;

    // initialization
    for(i=1; i<NC; i++) {
        c.coeff[i]=0;
        d.coeff[i]=0;
    }
    c.coeff[0] = 1;
    d.coeff[0] = 0;

    for(i=0; i<2*NC; i++) {
        btemp.coeff[i] = 0;           // initialize first
        if(i<NC) {
            f.coeff[i] = a->coeff[i];
            g.coeff[i] = p.coeff[i];
        } else {
            f.coeff[i] = 0;
            g.coeff[i] = 0;
        }
    }
```

```
    k = 0;
    int shiftcount;     //initialize
    // might need to be shifted
    // we shall assume that f != 0

    while(isOne(&f) == false) {
        //Now consider the normal case, where we want to shift a few bits
        cftemp = f.coeff[0];
        shiftcount  =  0;
        while ((cftemp&1) == 0) { //count the bits of evenness, up to 31
            shiftcount += 1;
            if(shiftcount == 31) break;    // can't shift more than 31
            cftemp = cftemp >> 1;
        }
        if (shiftcount >= 1) {      //the typical case
            for(i=0; i<NC+1; i++) { // f = f / 2^shiftcount, f is bigNum but <= 2p
                f.coeff[i] = f.coeff[i] >> shiftcount;
                if(i != NC) f.coeff[i] |= (f.coeff[i+1] << (NBIT - shiftcount));
            }
            k = k + shiftcount;

            if( isOne(&f) == true) break;       //We can break here without
            //changing d, since that does not affect c or k at this point
            //Now we want to shift and reduce d, if f != 0X01, and k>0


            btemp.coeff[NC] = 0;        // clear up btemp.coeff[NC] every time
            for(i=NC-1; i>=0; i--) {     // d = d << shiftcount
                btemp.coeff[i] = d.coeff[i] << shiftcount;
                btemp.coeff[i+1] |= (d.coeff[i] >> (NBIT - shiftcount));
            }
            // if btemp is greater than smallNum, then reduce mod p
            if(btemp.coeff[NC]!=0) {
                d = reduce(&btemp);
            } else {    // if btemp is not greater than smallNum
                for(i=0; i<NC; i++) d.coeff[i] = btemp.coeff[i];
            }
        }   //end if shiftcount >= 1

        // if f < g
        if(firstbig(&f, &g) == -1) {
            btemp = f; f = g; g = btemp;    // exchange f,g
            stemp = c; c = d; d = stemp;    // exchange c,d
        }

        // if f = g (mod 4)
        if(((f.coeff[0] ^ g.coeff[0]) & 3) == 0) {
            f = sub_big(&f, &g);        // f = f - g
            c = sub_and_reduce(&c, &d); // c = c - d
        } else {        // if f != g (mod 4)
            f = add_big(&f, &g);        // f = f + g
            c = add_and_reduce(&c, &d); // c = c + d
        }       //end if mod 4 (and else)
    }       //end while, outer loop

//  p*p = 1 mod 2^32, so r = 1;

    // ****** fixup algorithm *******
    for(i=0; i<NC; i++) {
        x.coeff[i] = c.coeff[i];    // x = c
        v.coeff[i] = 0; // initialize v
        ans.coeff[i] = 0;
        ans2.coeff[i] = 0;
```

```
    }
    for(i=NC; i<2*NC; i++) {
        x.coeff[i] = 0;
        ans.coeff[i] = 0;
        ans2.coeff[i] = 0;
    }

    while(k != 0) {
        if(k<32) j = k;
        else j = 32;

        v.coeff[0] = x.coeff[0] & table[j];

        ans.coeff[NC] = v.coeff[0];
        ans.coeff[NC-2] = v.coeff[0];
        ans.coeff[3] = v.coeff[0];
        ans2.coeff[NC-1] = v.coeff[0];
        ans2.coeff[0] = v.coeff[0];

        btemp = sub_big(&ans, &ans2);

        x = add_big(&btemp, &x);

        if(j == 32) {   // get rid of x.coeff[0] and shift bytes down
            for(i=0; i<=NC; i++) {
                x.coeff[i] = x.coeff[i+1];
            }
        } else {
            for(i=0; i<2*NC-1; i++) {
                x.coeff[i] >>= j;
                x.coeff[i] |= (x.coeff[i+1] << (32-j));
            }
        }
        k = k - j;
    }

    for(i=0; i<NC; i++) inv.coeff[i] = x.coeff[i];

    return inv; // x is the inverse of a (mod p)
}


int firstbig(smallNum *a, smallNum *b)
{
    for(int i=NC-1; i>=0; i--) {
        if(a->coeff[i] > b->coeff[i]) return  1;
        if(a->coeff[i] < b->coeff[i]) return -1;
    }
    return 0;
}


int firstbig(bigNum *a, bigNum *b)
{
    for(int i=2*NC-1; i>=0; i--) {
        if(a->coeff[i] > b->coeff[i]) return  1;
        if(a->coeff[i] < b->coeff[i]) return -1;
    }
    return 0;
}


// return c=a+b, all are bigNum
bigNum add_big(bigNum *a, bigNum *b)
```

```
{
    bigNum c;
    unsigned long long int at, bt, temp;
    int carry[2*NC+1];

    // initialization of carry and c
    for(int i=0; i<2*NC; i++) {
        c.coeff[i]=0;
        carry[i]=0;
    }
    carry[2*NC]=0;

    for(int i=0; i<2*NC; i++) {
        at = a->coeff[i];
        bt = b->coeff[i];
        temp = at + bt + carry[i];
        if(temp>cut) {          // temp should always be less than cut
            temp = temp - cut - 1;
            carry[i+1] = 1;     // carry over to the next coefficient
        }
        c.coeff[i] = temp;
    }

    return c;
}


// test whether a > b
bool compare(bigNum *a, bigNum *b)
{
    for(int i=2*NC-1; i>=0; i--) {
        if(a->coeff[i] != b->coeff[i]) return false;
    }

    return true;
}


// test whether a = 1
bool isOne(bigNum *a)
{
    for(int i=2*NC-1; i>0; i--) {
        if(a->coeff[i] != 0) return false;
    }
    if(a->coeff[0] == 1) return true;
    return false;
}


// return a-b
bigNum sub_big(bigNum *a, bigNum *b)
{
    bigNum c;
    unsigned long long int temp;
    int borrow[2*NC+1];


    // initialization of borrow and c
    for(int i=0; i<2*NC; i++) {
        c.coeff[i]=0;
        borrow[i]=0;
    }
    borrow[2*NC]=0;
```

```
    for(int i=0; i<2*NC; i++) {
        // to avoid 0-1 = 2^32-1 w/o borrowing
        if(a->coeff[i]==0 && borrow[i]==1) {
            a->coeff[i] = cut + 1;
            borrow[i+1] = 1;
        }
        if(a->coeff[i] - borrow[i] >= b->coeff[i]) {
            c.coeff[i] = a->coeff[i] - borrow[i] - b->coeff[i];
        } else {
            temp = a->coeff[i] + cut + 1;
            c.coeff[i] = temp - b->coeff[i] - borrow[i];
            borrow[i+1] = 1;
        }
    }

    return c;
}


// multiply but not reduce mod p
bigNum multiply_noreduce(smallNum *a, smallNum *b)
{
    bigNum c;
    unsigned long long int at, bt, ct;
    unsigned long long int temp;
    int carry[2*NC+1];

    // initialization
    for(int i=0; i<2*NC; i++) {
        c.coeff[i]=0;
        carry[i]=0;
    }

    for(int i=0; i<NC; i++) {
        for(int j=0; j<NC; j++) {
            at=a->coeff[i];
            bt=b->coeff[j];
            ct=at*bt;

            /*calculate c.coeff[i+j]*/
            temp=ct&cut;
            temp=temp+c.coeff[i+j]+carry[i+j];
            carry[i+j]=0;
            if(temp>cut) {
                carry[i+j+1] += 1;
                temp -= (cut+1);
            }
            c.coeff[i+j] = temp;

            /*calculate c.coeff[i+j+1]*/
            temp=ct>>32;
            temp=temp+c.coeff[i+j+1]+carry[i+j+1];
            carry[i+j+1]=0;
            if(temp>cut) {
                carry[i+j+2] += 1;
                temp -= (cut+1);
            }
            c.coeff[i+j+1] = temp;
        }
    }
    c.deg=find_degree(&c);

    return c;
}
```

Acknowledgements

We would like to thank Mr. Rich Schroeppel for the use of his algorithm, as well as his explanations of elliptic curve theory.

Thanks also to Mr. William Cordwell for mentoring our team and Mr. Stephen Schum for his encouragement and sponsorship of our project.

Finally, thank you to Sandia and Los Alamos National Labs for sponsoring the Supercomputing Challenge.