Problem Set 3 Solutions

Problem 3-1. Inverses

In general, there are two ways to compute $a^{-1} \bmod p$ where p is a prime and a is a random element of Z_p:

1. Use the equation $a^{-1} \bmod p = a^{p-2} \bmod p$.
2. Use Euclid's extended algorithm (see CLRS chapter 31).

Program both of these approaches and give the average times found for many such random a's. (You can let the p be the same.) Which is better? Summarize your results and submit your code.

We suspected that students would find different answers depending on the computer architecture, operating system, and programming language. We note that the GNU Privacy Guard uses Extended Euclid for computing inverses[1]. Knuth [1] has a long discussion on the difficulty of finding tight bounds on the running time of Extended Euclid in section 4.5.2. We have the upper bound on running time $O((\log a)^3)$ of bit operations on Euclid, but we do not have a tighter bound without making further assumptions.

The asymptotic running time of Extended Euclid and exponentiation should only differ by a constant factor. This constant factor will not likely manifest itself until the prime becomes sufficiently large. In all the solutions that tested large primes, Extended Euclid eventually performs better than exponentiation. The difference between the running times may also depend on caching of information between executions of the loop.

Of importance is using a big number package. Students who only looked at primes with fewer than 512 bits would usually not observe the point at which the running times of Extended Euclid and exponentiation cross.

Students were allowed to use a built-in modular exponentiation routine. But because the built-in routine is optimized compared to the hand-crafted Extended Euclid algorithm, the exponentiation routine will likely perform fast for small primes. As the length of the prime increases in your experiments, the Extended Euclid algorithm tended to perform better than exponentiation by a constant factor. Students who coded their own modular exponentiation routine tended not to notice this crossover because hand-crafted exponentiation ran slowly even for small primes.

It should be noted that when working on k-bit numbers, both procedures will do $O(k)$ multiplications of numbers of length $O(k)$, so one would expect that the running times should approach a fixed ratio as k gets large. (There are clever ways of speeding up Euclid, but this isn't what we were implementing here...)

To make a long story short, Extended Euclid and exponentiation have worst case asymptotic running times that differ only by a constant factor. Actual observations seem to agree that Extended Euclid is on average faster for this workload (large primes, many random a values, caching). The

[1] See `mpi/mpi-inv.c` in `ftp://ftp.pgpi.org/pub/pgp/gnupg/gnupg-1.0.6.tar.gz`.

standard textbook ways of implementing these two algorithms have the same asymptotic running times.

Petros Boufounos, Luciano Castagnola, and Nikos Michalakis submitted the following solution.

A1 Use the equation $a^{-1} \bmod p = a^{p-2} \bmod p$.

A2 Use Euclid's extended algorithm.

We tested the efficiency of the two algorithms in the following manner:

1. Select a random prime number p of bit-length l
2. Generate a test-set array of random values $a_i < p$
3. Run Java's BigInteger modPow method (A1) on the a_i and report the total time taken (in milliseconds).
4. Run our implementation of Euclid's extended algorithm (A2) on the a_i and report the total time taken (in milliseconds).

We did this several times, using different bit-lengths for the chosen p in order to compare the orders of growth of the two algorithms.

Table 1 and Figure 1 show that to answer the question of which method is better depends on the size of p. For small p (roughly below 500 bits) Java's modular exponentiation is faster than our implementation of Euclid's extended algorithm. But for large enough p, Euclid's extended algorithm runs faster.

Bit-length	Time A1 (ms)	Time A2 (ms)
64	42	469
128	148	903
192	441	1408
256	763	2099
320	1385	2574
384	2275	3200
448	3570	4096
512	4942	4600
576	6739	5279
640	9135	6049
704	12066	7097
768	15462	7700
832	19274	8489
896	23649	9424
1024	34891	11191

Table 1: Running times

The Java code is available on the 6.857 Web page.

Problem 3-2. ElGamal

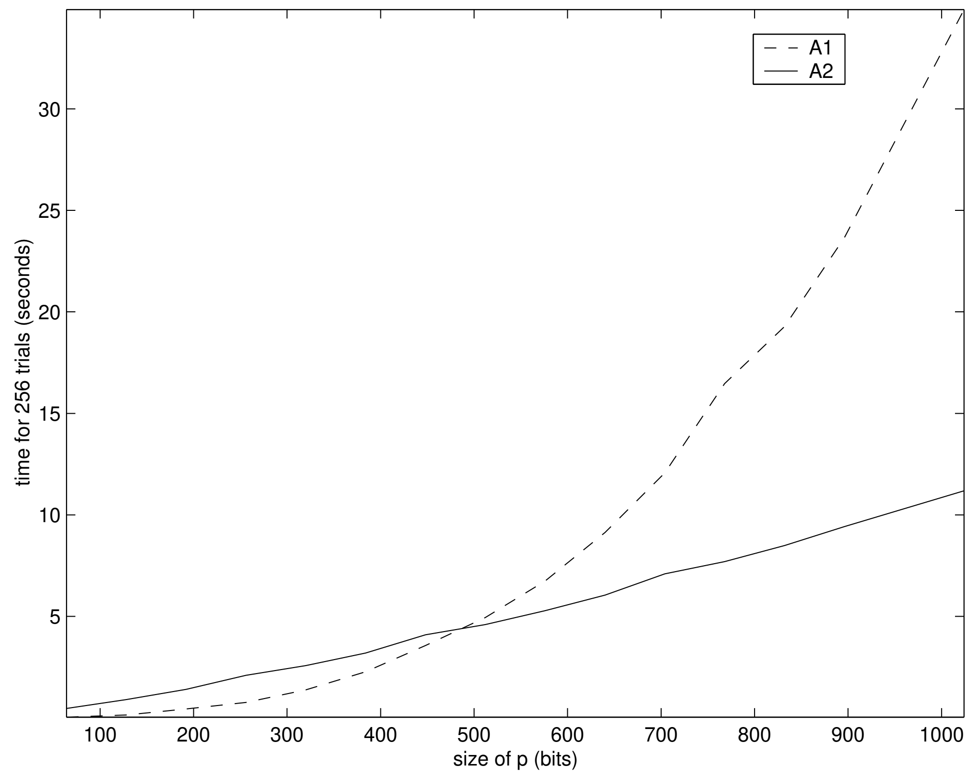

Figure 1: Extended Euclid performs better after reaching 512-bit primes. The staff suspects that the great slope differences are a result of caching.

Explain how to generalize the ElGamal signature scheme to work over 2×2 invertible matrices modulo p, where p is prime. (All elements we work with are then represented by such 2×2 invertible matrices instead of by individual elements modulo p.) The group of such matrices is typically denoted $\mathrm{GL}(2, p)$.

Hint: Let $f(p)$ denote the number of invertible 2×2 matrices modulo p. Argue that $f(p) = (p^2 - 1)(p^2 - p)$. Note that all elements have an order that divides $f(p)$ since $f(p)$ is the size of the group. But the group may not have a generator, so you might need an element of a large order instead.

There are trivial matrices that generate a subgroup of size $p - 1$. One such matrix is

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \bmod p$$

where a and b are each order $(p - 1)$. We gave partial credit for solutions of this manner. Students received more points for exploring the relationship between the order of a matrix and $f(p)$. For instance, one could craft a prime p which reduces the number of factors in $f(p)$ which in turn reduces the number of possibilities for the order of any given matrix element.

It was also necessary to demonstrate that a valid signature still verifies correctly. Some students used a hash function to produce scalars from matrices without realizing that the signature verification no longer works. This is the right direction, but not quite right.

Cristian Cadar, Cătălin Francu, and Ovidiu Gheorghioiu submitted this solution:

Notation: We name all the 2×2 matrices using bold capital letters, as in **G, R, Y,** $\ldots$. We name all the numeric variables using italic letters, as in x, r, s, $\ldots$.

Claim: The number of non-invertible 2×2 matrices modulo p is $p^3 + p^2 - p$, if p is prime.

Proof: Let $\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ be a 2×2 matrix. **A** is non-invertible if and only if $ad \equiv bc \pmod{p}$. There are two cases:

1. $ad \equiv bc \equiv 0 \pmod{p}$. Because p is prime, either a or d must be 0 (yielding $2p-1$ possible combinations for a and d), and either b or c must be 0 (yielding $2p-1$ possible combinations for b and c). Therefore there are $(2p-1)^2$ ways that $ad \equiv bc \equiv 0 \pmod{p}$.
2. $ad \equiv bc \not\equiv 0 \pmod{p}$. The multiplication table modulo p contains the number 0 exactly $2p-1$ times, and each of the numbers $1, 2, \ldots, p-1$ exactly $p-1$ times. Hence, there exist $(p-1)^2$ combinations of a and d such that $ad \neq 0$, and for each of them there exist $p-1$ combinations of b and c such that $bc \equiv ad \pmod{p}$. Therefore, there exist $(p-1)^3$ ways that $ad \equiv bc \not\equiv 0 \pmod{p}$.

The total number of non-invertible matrices is

$$(2p-1)^2 + (p-1)^3 = p^3 + p^2 - p$$

Corollary: $|\mathrm{GL}(2,p)| = f(p) = p^4 - (p^3 + p^2 - p) = (p-1)^2 p(p+1)$

Let us assume that p is a 1024-bit number. We need a good bijection between the 4096-bit numbers and the 2×2 matrices with 1024-bit elements. Concatenation of the 4 elements isn't a viable solution because several matrices can map to the same 4096-bit number, but such a bijection shouldn't be too hard to find. With a little effort, we could even enumerate the elements of $\mathrm{GL}(2,p)$ in lexicographical order. Let

$$d : \mathrm{GL}(2, \mathrm{p}) \to \mathbf{N}$$

be a public function that maps matrices to numbers.

The next step is to find a good p such that a generator modulo $f(p)$ is reasonably easy to find. We want to find triplets of primes of the form:

$$\begin{aligned} p &= 12k + 7 \\ p_1 &= 3k + 2 \\ p_2 &= 2k + 1 \end{aligned}$$

This would insure that:

$$\begin{aligned} f(p) &= (p-1)^2 p(p+1) \\ &= (6p_2)^2 p(4p_1) \\ &= 2^4 3^2 p_2^2 p p_1 \end{aligned}$$

Now if a matrix **G** has order greater than 3, the order is likely to be large. A quick program (found on the 6.857 Web page) revealed that such triplets are in fact abundant. The rest of the signature scheme is merely an extension of ElGamal:

Keygen: Generate a triplet (p, p_1, p_2) as above (1024 bits)
Find a generator matrix $\mathbf{G}$ of GL(2,p)
Choose $x \in_R \{0, 1, \ldots, f(p)-1\}$
Compute $\mathbf{Y} = \mathbf{G}^x \pmod{p}$
(meaning that each of the elements of $\mathbf{Y}$ is computed modulo p)
PK = (p, $\mathbf{G}$, $\mathbf{Y}$)
SK = (x);

Sign(M): $m = h(M)$ (where h is a collision-resistant hash function)
Choose $k \in_R \{1, \ldots, f(p)-1\}$ s.t. $\gcd(k, f(p)) = 1$
$\mathbf{R} = \mathbf{G}^k \pmod{p}$
$r = d(\mathbf{R})$
$s = (m - rx) \cdot k^{-1} \pmod{f(p)}$
Output $\sigma = (\mathbf{R}, \mathbf{s})$

Verify(M, σ, PK): $m = h(M)$
Compute $r = d(\mathbf{R})$
Check that $0 < r < f(p)$
Check that $\mathbf{Y^r R^s} \equiv \mathbf{G^m} \mod \mathbf{p}$

This works because

$$\mathbf{Y^r R^s} \equiv \mathbf{G^{xr} G^{ks}} \equiv \mathbf{G^m} \pmod{\mathbf{p}}$$

Problem 3-3. Blind signatures

Consider the following well-known signature scheme presented below. (Assume that $H()$ is a collision-resistant hash function.)

Key Generation
Find two primes, p, q such that $q|p-1$.
Find elements $g, h \in Z_p^*$ of order q.
(p, q, g, h) are global parameters
Pick $r, s \in Z_q$
Public key is $(y = g^{-r}h^{-s})$
Secret key is (r, s)

Sign(m)
Pick $t, u \in Z_q$
Let $a = g^t h^u \bmod p$
Let $c = H(m, a)$
Let $R = t + cr \bmod q$
Let $S = u + cs \bmod q$
Output (a, R, S)

In order to verify this signature, first compute $c = H(m, a)$ and then check whether $a = g^R h^S y^c \bmod p$. This works because

$$\begin{aligned} g^R h^S y^c &= g^{t+cr} h^{u+cs} y^c \\ &= g^{t+cr} h^{u+cs} (g^{-r} h^{-s})^c \\ &= g^t h^u = a \end{aligned}$$

Verification Ben Bitdiddle wants to modify this scheme into a *blind* signature scheme. In this setup, there is an Authority that holds a secret key and has the power to sign messages. A user would like to have the Authority sign a message m in such a way that the Authority learns nothing about either m or sign(m). Nonetheless, any outsider can verify that sign(m) is a signature of m made by the Authority. One could imagine this kind of application in a notary public that notarizes only sealed envelopes, in a voting scheme, or in an electronic cash systems.

In the original signature scheme, the signer knows m and computes $c = H(m, a)$. Instead of this, Ben envisions the following:

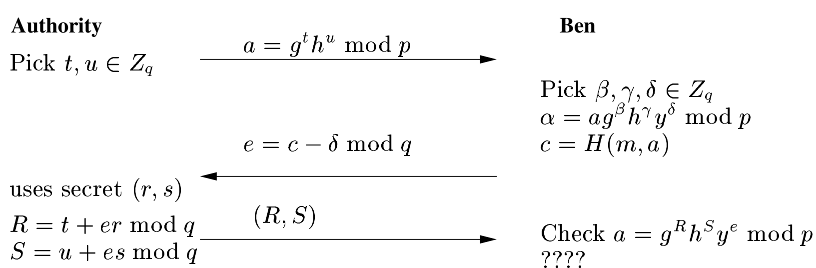

Figure 2: Ben's Blind Signature Scheme

1. The Authority picks $t, u \in Z_q$, and sends Ben the value $a = g^t h^u \bmod p$.
2. Ben picks $\beta, \gamma, \delta \in Z_q$ and blinds a into $\alpha = a g^\beta h^\gamma y^\delta \bmod p$. Ben also computes $c = H(m, \alpha)$ and sends $e = c - \delta \bmod q$ to the Authority.
3. Using the secret key (r, s), the Authority computes $R = t + er \bmod q$, $S = u + es \bmod q$ and sends (R, S) to Ben.
4. Ben makes sure that $a = g^R h^S y^e$ as before in the original scheme. Ben derives a signature for m given R, S.

Jeremy Mineweaser, Benjamin Pick, Christopher Stein, and Omri Traub gave us this solution to part (a) and (b). The answer to part(c) is inspired by an email from David Pointcheval, a noted cryptographer and author of a paper in which this blind signature scheme (Okamoto-Schnorr) is published.

(a) Help Ben complete this signature scheme by describing exactly what Ben should publish in step (4). At this point, Ben knows $(a, \alpha, \beta, \gamma, \delta, c, e, R, S)$. However, since the signature should be blind, it cannot simply be (a, R, S) as before. Be sure to explain what a verifier needs to check in order to make sure that the signature is correct.

Solution:

In step 4 of the signature scheme, Ben should publish $\sigma = (\alpha, \beta + R, \gamma + S)$ as the blind signature for the message m. In this way, anyone can use σ in conjunction with the Authority's public key, y, to verify the correctness of the signature. Ben keeps secret the random values β, γ, and δ; this prevents the Authority from linking any observed σ to a particular instance of the signing process.

The task of the verifier is to show that the message m was signed by the Authority whose public key is y. Verification begins with the message m and its signature σ. First, m and α are concatenated, then hashed with the collision-resistant hash function, H, to produce c. That is, $c = H(m, \alpha)$. The global parameters g, h, and p are then used to confirm that $\alpha = g^{\beta+R} h^{\gamma+S} y^c \bmod p$.

If this holds, then the signature is valid. The following steps show how the above equation corresponds to the computation that Ben performed during the signing process to obtain α.

$$\begin{aligned}
g^{\beta+R} h^{\gamma+S} y^c &= g^{\beta+R} h^{\gamma+S} (g^{-r} h^{-s})^c \\
&= g^{\beta+R} h^{\gamma+S} (g^{-cr} h^{-sc}) \\
&= g^{\beta+R-cr} h^{\gamma+S-sc} \\
&= g^{\beta+(t+er)-cr} h^{\gamma+(u+es)-sc} \\
&= g^{\beta+(t-r\delta+cr)-cr} h^{\gamma+(u-s\delta+sc)-sc} \\
&= g^{\beta-r\delta+t} h^{\gamma-s\delta+u} \\
&= a g^{\beta-r\delta} h^{\gamma-s\delta} \\
&= a g^{\beta} h^{\gamma} y^{\delta} \\
&= \alpha
\end{aligned}$$

(b) Argue that the Authority learns nothing about either m or sign(m).

Solution:

The blind signature scheme proceeds in three steps. First the Authority computes a and sends it to Ben. Ben replies with e; the Authority signs e and returns R and S. At no time during or after these steps does the Authority learn of m or $sign(m)$, σ. This is because β, γ, and δ are not revealed during the signing process.

m enters into the process as part of the input to the hash function, H. None of m, α, and $c = H(m, \alpha)$ is known to the Authority, since the secret random value δ blinds c into e for signing by the Authority.

To learn m from e, the Authority would need to know both α and δ and be able to invert H to obtain (m, α) from $c = e + \delta \bmod q$.

$sign(m)$ is not revealed by Ben to the Authority during the signing process. Neither can $sign(m)$ be derived from the values known to the Authority, which are a, e, R, and S. Since β, γ, and δ are unknown (to the Authority), none of the components of the signature can be determined. [or rather, each of the components of the signature are uniformly distributed among the elements of the group].

[**Editor's Note:** Intuitively, the answer above is arguing that α, ρ, σ are uniformly distributed and so the Authority can learn nothing about the signature.

A good way to argue this more formally might be to reason as follows. Suppose the Authority has a way to learn either m or $sign(m)$ from the transcript with the User. Well perhaps then, we can trick the Authority to interact with a Simulator of the user instead of the actual user. The Simulator is just a program that does not know the m or $sign(m)$, but does have access to random bits. If we can then argue that the Authority cannot tell the difference between interacting with the Simulator and interacting with the User herself, then we can argue that the Authority never learns anything he could not already figure out himself by just playing with the Simulator.

In order to do this, we note that the only way the Authority can tell the difference between a Simulator and an actual User is through a property of the transcripts that the Authority sees. We can therefore use the analysis above to show how transcripts from a real User are distributed uniformly.]

(c) Argue that the Authority has no way of linking the values that she sees with either m or $\text{sign}(m)$. In other words, suppose the Authority cleverly chooses the values (a, R, S) and remembers them. Suppose later she sees a signature $m, (\alpha, \rho, \sigma)$. Is there a way for her to use her database of remembered values to learn anything about who submitted the message or when it was signed?

Solution:

Lets suppose the Authority maintains a database of

$$[\text{User}_i, a_i, e_i, R_i, S_i]$$

for each signing interaction. Now suppose the Authority observes $(m, \text{sign}(m) = (\alpha, \rho, \sigma))$ and wishes to determine when and/or by whom the message was submitted for signing.

The problem is that for any entry $[\text{User}_i, a_i, R_i, S_i]$ in the database, there are suitable values for β, γ, δ that would give rise to this pair of $m, \text{sign}(m)$. If these values are chosen well, the Authority cannot link the signature with a database entry.

Two groups of students proposed the following clever attack:

1. On input $(m, (\alpha, \rho, \sigma))$,
2. Compute $c = H(m, \alpha)$ based on the observed values.
3. For each entry, i of the database, test whether $\alpha = a_i g^{\rho - R_i} h^{\sigma - S_i} y^{c - e_i}$
4. If so, output "User i signed this message" otherwise continue with the next entry in the database.

However, David Pointcheval points out how this attack does not work. First, note that all entries in the database satisfy $a_i = g^{R_i} h^{S_i} y^{e_i}$. In addition, the valid signature satisfies $\alpha = g^{\rho} h^{\sigma} y^{c}$ where $c = H(m, \alpha)$. Therefore, all entries in the database also satisfy: $\alpha / a_i = g^{\rho - R_i} h^{\sigma - S_i} y^{c - e_i}$ and so the attack outputs all previous signers!

Note however, that neither of the above arguments are rigorous proofs of unlinkability.

References

[1] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1969. Second edition, 1981.