Handout 8: Problem Set #1 Solutions

Problem 1

Suppose p is a prime and g and h are both generators of Z_p^*. Prove or disprove the following statements:

$$\begin{array}{ll}
\textbf{A:} & \{x \leftarrow Z_p^* : g^x \mod p\} = \{x \leftarrow Z_p^*; y \leftarrow Z_p^* : g^{xy} \mod p\} \\
\textbf{B:} & \{x \leftarrow Z_p^* : g^x \mod p\} = \{x \leftarrow Z_p^* : h^x \mod p\} \\
\textbf{C:} & \{x \leftarrow Z_p^* : g^x \mod p\} = \{x \leftarrow Z_p^* : x^g \mod p\} \\
\textbf{D:} & \{x \leftarrow Z_p^* : x^g \mod p\} = \{x \leftarrow Z_p^* : x^{gh} \mod p\}
\end{array}$$

(Recall from Handout #3 that $\{x \leftarrow Z_p^* : g^x \mod p\}$ is a probability distribution. You are being asked to prove or disprove the statement that two probability distributions are identically distributed.)

Solution:

A: Not equal. The left distribution is uniform over Z_p^* (See Part B). Therefore, on the left, g^x is a quadratic residue with probability 1/2. In the right distribution, with probability 3/4, either x or y is even. Therefore, g^{xy} is a quadratic residue with probability 3/4. Thus the two distributions cannot be equal.

B: Equal. Because g and h are generators, the maps $x \mapsto g^x$ and $x \mapsto h^x$ are bijective from Z_p^* to Z_p^*. Therefore both distributions are uniform over Z_p^*.

C: Not equal. Let $p = 3, g = 2$. Then the left distribution is uniform over Z_p^*, while the right distribution has probability 1 on element 1.

D: Not equal. Let $p = 5, g = 2, h = 2$ (note that g and h need not be distinct). Then the left distribution is uniform over $\{1, -1\}$, while the right distribution has probability 1 on element 1.

Problem 2

Suppose that the Prime Discrete Logarithm Problem is easy. That is, suppose that there exists a probabilistic, polynomial time algorithm A that, on inputs p, g and $g^x \mod p$, outputs x if p is a prime, g is a generator of Z_p^* and $g^x \mod p$ is prime. Show that there exists a probabilistic polynomial-time algorithm, B, that solves the Discrete Logarithm Problem.

Solution:

The main idea here is to use the idea of random self-reducibility. That is, we want to reduce solving the discrete logarithm problem for a particular g^x to solving the discrete logarithm problem for a uniformly random input g^{y+x}. Then since a uniformly random input is likely to be prime, we can apply our algorithm for the Prime Discrete Logarithm Problem to g^{x+y}.

Let A be an algorithm for solving the Prime Discrete Logarithm Problem. Our reduction algorithm, B, works as follows: "on input (p, g, g^x), pick a random $y \leftarrow Z_p^*$, and check if $g^x g^y \bmod p$ is prime. If not, choose a new y until that condition is satisfied. Then pass $(p, g, g^x g^y \bmod p)$ to A, and receive from it a value z. Return $z - y$."

First we prove that B is PPT: this amounts to analyzing how many ys we must choose before $g^x g^y \bmod p$ is prime. Note that $g^x g^y \bmod p$ is a uniformly random element of Z_p^*, and by the prime number theorem, an $\Omega(1/\log p)$ fraction of those elements are prime. Therefore we expect to choose $O(\log p)$ such y. Because A is poly-time, B is expected poly-time.

The correctness of the algorithm is clear. Since $g^x g^y \bmod p = g^{x+y} \bmod p$, the probability that our algorithm returns x is equal to the probability that A returns $x + y$ when g^{xy} is a uniformly random prime number less than p.

Problem 3

We define the Lily problem as: given two integers n and S determine whether S is relatively prime to $\phi(n)$. Prove that if it is hard to determine on inputs two integers n and e whether e is relatively prime with $\phi(n)$, then the RSA function is hard to invert.

Solution:

The main idea here is that if n and e are relatively prime then $f(x) = x^e \bmod n$ is a permutation, but if n and e are not relatively prime then $f(x)$ is a many-to-one mapping. Therefore, if we choose x at random, an RSA inverting algorithm cannot return x on input x^e with probability better than 1/2. Our reduction will show that we can solve the Lily problem with error probability 1/2 for any n and e such that RSA is "easy" for that n and e. (Note that we could repeat our procedure many times to reduce the error probability.)

Suppose for contradiction that RSA is "easy" to invert. That is, there exists a PPT A such that given (n, e, c) where n is an integer, e is relatively prime with $\phi(n)$ and $m \in Z_n^*$, $A(n, e, m^e)$ outputs m such that $m^e = c \bmod n$. For simplicity, we will assume that our RSA inverter inverts with probability 1 over the choice of m. (If the RSA inverter sometimes failed, we could use a random self-reducibility argument to create an RSA inverter that works with overwhelming probability over the choice of m.)

We construct a B which solves the Lily problem as follows: "on input (n, e), choose $m \leftarrow Z_n^*$ at random and give $(n, e, m^e \bmod n)$ to A. If A returns m then output `Relatively Prime` and otherwise output `Not Relatively Prime`.

B is clearly PPT since A is PPT. Now, if $(e, \phi(n)) = 1$, then our RSA inverter is receiving a valid input $(n, e, m^e \bmod n)$ and is obligated to output m in which case B correctly outputs `Relatively Prime`.

Now suppose $(e, \phi(n)) > 1$. We claim that every eth residue mod$n$ has at least two eth roots (the proof is given below). Since m is chosen randomly, A has absolutely no information about which of the eth roots of m^e is the m we started with. Therefore, no matter how A answers in this case, it cannot cause us to output `Relatively Prime` with probability greater than $1/2$.

Finally, we prove the claim that when $(e, \phi(n)) = \alpha$, every m^e has at least two eth roots mod n. Let $\beta \neq 1$ be an element such that $\beta^e = 1$. (By Cauchy's Theorem such an element must exist.) Then $\beta m \neq m \bmod n$ but $(\beta m)^e = \beta^e m^e = m^e \bmod n$. Thus m and βm are distinct eth roots of $m^e \bmod n$.

Problem 4: Factoring

Let O_n be an oracle that on input x returns a square root of $x \mod n$, if one exists, and $\perp$ otherwise. Prove that there exists a probabilistic polynomial-time algorithm that on input an integer n and access to O_n outputs n's factorization.

Solution:

Recall that in class we proved a similar result when n is a product of two distinct odd primes. The exact same technique yields that if we can take square roots mod n then we can find a non-trivial factor α of n. We would like to recurse on α and n/α but this would require taking square roots mod α and n/α and we have only an oracle for square roots mod n. Therefore, we show that our oracle for square roots mod n can be used to find square roots mod d for any d dividing n.

First we will outline our algorithm, then we will fill in the details. On input d, (where d

divides n) algorithm A does the following:

1. If 2 divides d then store 2 and run $A(d/2)$.
2. If d is a prime power p^k then store p^k and halt.
3. Choose x at random from Z_d^* and find a square root $y \neq x$ and $y \neq -x$ of x^2 mod n.
4. Let $\alpha = gcd(y + x, n)$.
5. Run $A(\alpha)$ and $A(n/\alpha)$.

In Step 2, observe that if d is a prime power then k is at most $log(d)$. Therefore, for each $i < log(d)$ we can take the ith root of d and test whether this root is prime. (ith roots can be found in many ways, in particular using Newton's Method.) This allows us to determine in polynomial time whether d is a prime power.

In Step 4, observe that an argument identical to what we saw in class (when we considered the special case where n was the product of two primes) yields that α is a non-trivial factor of n.

All that remains is to handle Step 4. Here we must use O_n to find a square root y of $x^2 \mod d$ where $y \neq x$ and $y \neq -x$. Here we observe that if $y^2 = x^2$ mod n (that is, that y is a square root of x^2 mod n) then n divides $y^2 - x^2$. Therefore, since d divides n, d divides $y^2 - x^2$. This implies that $y^2 = x^2 \mod d$. We have thus shown that if y is a square root of x^2 mod n then y is a square root of x^2 mod d. Thus $O_n(x^2)$ returns a square root of x mod d. All that remains is to ensure $y \neq x$ mod d and $y \neq -x$ mod d. Here we observe that x^2 has at least 4 square roots mod d (since d is odd and not a prime power). O_n has absolutely no information about which of the square roots of x^2 we started with. Therefore, O_n must give us a square root y not equal to plus or minus x mod d at least half the time. Thus we can just repeat Step 3 a small number of times and we are very likely to get x and y with the desired properties.