Algebraic Cryptography
Exam 1 Solutions

Choose 5 of the following 7 problems to complete.

Problem 1. Recall that in the RSA cryptosystem a user generates primes p, q and computes the product n. The user then generates an encryption exponent e relatively prime to $(p-1)$ and $(q-1)$. Then the user computes a decryption exponent d so that $de \equiv 1 \pmod{(p-1)(q-1)}$. The user then publishes the public key (n, e) and keeps the private key d a secret.

(a) Prove that the map $x \mapsto x^d \pmod{n}$ is the inverse of $x \mapsto x^e \pmod{n}$.

(b) Suppose Alice and Bob generate public keys (n, e) and (n', e') which share a prime p (i.e., $n = pq$ and $n' = pq'$), perhaps due to insufficient entropy (randomness) during key generation. Explain how an attacker, with knowledge only of the public keys can crack both Alice's and Bob's private keys.

Proof. (a) Since $m^{ed} = (m^e)^d$, we are ultimately trying to prove that $m^{ed} \equiv m \pmod{n}$ for all m. We first prove that $m^{ed} \equiv m \pmod{p}$ (and similarly for q). If $m \equiv 0 \pmod{p}$, then the result is trivial. Otherwise, notice that $de = 1 + k(p-1)(q-1)$ for some integer k. Then

$$m^{ed} = m^{1+k(p-1)(q-1)} = m \cdot (m^{(p-1)})^{k(q-1)} \equiv m \cdot 1^{k(q-1)} \pmod{p} \equiv m \pmod{p},$$

where the first congruence is due to Fermat's Little Theorem. A symmetric argument proves $m^{ed} \equiv m \pmod{q}$.

The previous paragraph proves $m^{ed} \equiv m \pmod{p}$ and $m^{ed} \equiv m \pmod{q}$. Since p, q are distinct primes, they are relatively prime, so we may apply the Chinese Remainder Theorem to conclude $m^{ed} \equiv m \pmod{pq} = m \pmod{n}$.

(b) Suppose Alice and Bob have public keys (n, e) and (n', e') which share a prime p (i.e., $n = pq$ and $n' = pq'$). Since n, n' are public, an attacker can compute $\gcd(n, n')$, yielding p, using the Euclidean Algorithm which is polynomial time (even of low degree). Then the attacker computes $\frac{n}{p}$ and $\frac{n'}{p}$,yielding q, q', and integer division is also polynomial time. Finally, the attacker solves for d, d' satisfying $de \equiv 1 \pmod{(p-1)(q-1)}$ and $d'e' \equiv 1 \pmod{(p-1)(q'-1)}$, again using the Euclidean Algorithm. Therefore the attacker has cracked the private keys, d, d' of Alice and Bob using only public information.

■

Problem 2. Consider a commutative public key encryption decryption scheme (i.e., the encryption functions e and d commute, $e \circ d = d \circ e$). Let e_A, d_A be Alice's encryption/decryption functions, and let e_B, d_B be Bob's. Let h denote a hash function and m a message. The functions e_A, e_B, h are all public knowledge.

(a) Explain how Alice can use these functions to sign a message m in such a way that everyone (not just Bob) can read the message and everyone can verify that Alice wrote it.

(b) Explain how Alice can send a secret message to Bob that only he can read, but that Bob (and only Bob) can know and be sure that Alice wrote it.

Proof. (a) Alice first hashes her message m creating $h(m)$. She then publishes her message m and appends her signature $d_A(h(m))$ using her private key function d_A. Obviously, anyone can read the message m, but they can also take her signature and apply her public key e_A to get $e_A(d_A(h(m))) = h(m)$. Finally, the recipient computes the hash $h(m)$ using the message they received, and verifies that these values match. Since only Alice knows d_A, the signature must have been created by Alice. Moreover, the message cannot have been tampered with (otherwise the computed hash wouldn't match the one Alice sent in her signature).

(b) Alice sends Bob an encrypted message using his public key $e_B(m)$. Obviously, anyone could send this, so Alice sends along the signature $e_B(d_A(h(m)))$. Only Bob can apply his private key to get $d_B(e_B(m)) = m$ and $d_B(e_B(d_A(h(m)))) = d_A(h(m))$ thus putting him in the situation of part (a).

■

Problem 3. Show that at least $\frac{n}{2}$ of the numbers $1, 2, \ldots, n$ have (binary) length equal to or greater than $\log_2 n - 1$. Then show that $n!$ has length at least equal to $\frac{n}{2}(\log_2 n - 2)$, and that for large n, this is greater than $Cn \ln n$ for some positive constant C.

Proof. Note that at least half of n is $\lceil \frac{n}{2} \rceil$. The numbers $\lfloor \frac{n}{2} \rfloor + 1, \ldots, n$ are exactly $\lceil \frac{n}{2} \rceil$ integers. Moreover, all of these are greater than or equal to $\frac{n}{2}$. Thus, all of these have length at least

$$\left\lfloor \log_2 \frac{n}{2} \right\rfloor + 1 = \lfloor \log_2 n - 1 \rfloor + 1 \geq \log_2 n - 2 + 1 = \log_2 n - 1$$

Recall that when multiplying numbers of length k, l, the product has length either, $k + l - 1$ or $k + l$, so at least $k + l - 1$. Now, we have $\lfloor \frac{n}{2} \rfloor$ multiplications to compute when computing the product: $n(n - 1) \cdots (\lfloor \frac{n}{2} \rfloor + 1)$. Since each number in this product has length at least $\log_2 n - 1$, the entire product has length at least

$$\overbrace{\left\lceil \frac{n}{2} \right\rceil (\log_2 n - 1)}^{\text{sum of lengths}} - \overbrace{\left\lfloor \frac{n}{2} \right\rfloor}^{\text{\# mults}} \geq \frac{n}{2}(\log_2 n - 1) - \frac{n}{2} = \frac{n}{2}(\log_2 n - 2).$$

Finally, since $\log_2 n = \frac{\ln n}{\ln 2}$, we consider the limit

$$\lim_{n \to \infty} \frac{\frac{n}{2}(\log_2 n - 2)}{n \ln n} = \lim_{n \to \infty} \left(\frac{n \ln n}{(2 \ln 2) n \ln n} - \frac{n}{n \ln n} \right) = \lim_{n \to \infty} \left(\frac{1}{2 \ln 2} - \frac{1}{\ln n} \right) = \frac{1}{2 \ln 2}.$$

Therefore, for every $\epsilon > 0$, there is an N such that for all $n \geq N$,

$$\frac{n}{2}(\log_2 n - 2) \geq \left(\frac{1}{2 \ln 2} - \epsilon \right) n \ln n.$$

In particular, if we set $\epsilon = \frac{1}{4 \ln 2}$, then $N = 16$, and so we may choose $C = \frac{1}{4 \ln 2}$. This proves that the length of $n!$ is $\Omega(n \ln n)$. ■

Problem 4. Given a k-bit integer, you want to compute the highest power of this number that has l or fewer bits (we suppose $l \gg k$). Estimate (with big-O) the number of bit operations required to do this. Your answer should be a very simple expression in terms of k and/or l. Moreover, you should explicitly describe the algorithm you are using.

Proof. Let n denote our k-bit integer. Note that the length of n^m is at least $km - (m - 1)$ and at most km. Therefore m is the largest exponent so that n^m has l bits, then m is at least $\frac{l}{k}$ and at most $\frac{l-1}{k-1}$. By a result from the homework (which you should reproduce for this solution), the number of bit operations to compute n^m is $O(m^2 \ln^2 n)$, since $\ln^2 n$ is proportional to k^2, the number of bit operations to compute n^m is

$$O\left(\left(\frac{l-1}{k-1} \right)^2 k^2 \right) = O(l^2)$$

since we are only concerned with large k, l (of course, still $l \gg k$).

By the way, in case you are wondering how to find m exactly, do it like this: Compute $n, n^2, n^{2^2}, n^{2^3}, \ldots$ until the value exceeds l bits. Then take the largest one of these that didn't exceed l bits, and multiply it by the next biggest. If it still doesn't exceed l bits, take the resulting product and repeat with the next largest term. If this product does exceed l bits, go back to the one that didn't and repeat with the next largest term. Eventually, you get a product of powers of n^{2^s}, and the sum of the 2^s terms that ended up in your product is m. (Note, I think this method might actually give you complexity $O(l^2 \ln \frac{l}{k})$ because we are multiplying the biggest terms first, instead of the smallest ones). ■

Problem 5. Suppose that you have a list of all primes having k or fewer bits. Using the Prime Number Theorem and big-O notation, estimate the number of bit operations needed to compute the sum of all these primes. You should explicitly describe the algorithm you are using. (*recall:* the number of bit operations required to add a k-bit number and an l-bit number is $\max\{k, l\}$.)

Proof. The primes that have k or fewer bits are those primes less than 2^k. By the Prime Number Theorem there are approximately $\pi(2^k) \asymp \frac{2^k}{k \ln 2}$ such primes. We will assume all of these primes have length k (this will be an overestimate, but not a significant one). To sum a large list of numbers all, we will employ the following strategy. Pair the numbers off and compute the sum of each pair, thus producing a new list of numbers (if the numbers of numbers is odd, just leave one alone). Repeat the previous step until we have the full sum. The number of steps in this algorithm is at most $\lceil \log_2 m \rceil$ where m is the number of integers we are adding. Moreover, if the numbers on one step all have length at most l, then the numbers on the next step all have length at most $l + 1$. So, let's assume we are going to sum m numbers all of length k. Then the sum of any pair on step j takes at most $O(k + j - 1)$ bit operations, and there are about $m2^{-j}$ such pairs. Thus, step j takes at most $O((k + j - 1)m2^{-j})$ bit operations. Summing this from $1 \leq j \leq \lceil \log_2 m \rceil$ gives the total time complexity. In our case, $m \asymp \frac{2^k}{k \ln 2}$, and hence $\lceil \log_2 m \rceil \asymp k$. Therefore, summing all the primes of length less than or equal to k has time complexity at most

$$\sum_{j=1}^{k} \frac{2^{k-j}}{k \ln 2}(j + k - 1) \leq \frac{1}{\ln 2} \sum_{j=1}^{k} 2^{k-j+1} \leq \frac{2^{k+1}}{\ln 2} = O(2^k).$$ ■

Problem 6. Let $\mathcal{P}_1$ be the decision problem

> **Input:** A polynomial $p(x)$ with integer coefficients.
> **Question:** Is there any interval of $\mathbb{R}$ on which $p(x)$ decreases?

Let $\mathcal{P}_2$ be the decision problem

> **Input:** A polynomial $p(x)$ with integer coefficients.
> **Question:** Is there any interval of $\mathbb{R}$ on which $p(x)$ is negative?

Show that $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent ($\mathcal{P}_1$ reduces to $\mathcal{P}_2$ and $\mathcal{P}_2$ reduces to $\mathcal{P}_1$).

Proof. Let $p(x)$ be a polynomial with integer coefficients. Notice that $p'(x)$ is also a polynomial with integer coefficients (because the coefficients are just the old integer coefficients multiplied by the integer powers). Moreover, $p'(x) < 0$ on some interval if and only if $p(x)$ is decreasing on that same interval. Therefore, given an instance of $\mathcal{P}_1$, we have constructed an instance of $\mathcal{P}_2$ which yields the same result. Thus we have reduce $\mathcal{P}_1$ to $\mathcal{P}_2$.

For the other direction, consider $p(x)$ and $\int p(x)\,dx$. Even though the former has integer coefficients, the latter may not (because the coefficients get divided by the powers when we integrate). Thus, if n denotes the degree of $p(x)$, we instead consider the polynomial $n! \int p(x)\,dx$ which *does* have integer coefficients (technically, we could replace $n!$ with the least common multiple of $1, \ldots, n$, but that isn't really easier). Moreover, $n! \int p(x)\,dx$ is decreasing if and only if $\int p(x)\,dx$ is decreasing if and only if $p(x) < 0$. Therefore, given an instance of $\mathcal{P}_2$, we have constructed an instance of $\mathcal{P}_1$ which yields the same result. Thus $\mathcal{P}_2$ reduces to $\mathcal{P}_1$.

Note: if we consider the degree of $p(x)$ as the input size we have *not* shown that $\mathcal{P}_2$ reduces to $\mathcal{P}_1$ in polynomial time because our algorithm requires the computation of $n!$, for which we do not have a polynomial time algorithm. ■

Problem 7. The Integer Factorization Search (IFS) Problem is the search problem

Input: An integer $N > 1$
Output: The statement "N is prime" or a nontrivial factor n of N.

The Integer Factorization Decision (IFD) Problem is the decision problem

Input: An integer $N > 1$ and an integer k
Question: Does N have a factor in the interval $[2, k]$?

Show that IFS reduces to IFD in polynomial time. That is, show there is a polynomial time algorithm (where the input size is the length of N) for IFS which makes at most polynomially many calls to an IFD-oracle.

Small bonus: Explain why IFD is in NP.

Proof. The solution is to apply binary search. The idea is to find the smallest factor of N (this factor will necessarily be prime, but that is not important). We first consider the interval $[a, b]$ where $a = 2, b = N - 1$ and query IFD with N and $k = N - 1$. If "no" then N is prime. If "yes" then N has a factor in the interval $[a, b]$. Now, while $a < b$, repeat the following loop:

- Query IFD with N and $k = \left\lceil \frac{a+b}{2} \right\rceil$.
- If "yes", set $b = \left\lceil \frac{a+b}{2} \right\rceil$.
- If "no", set $a = \left\lceil \frac{a+b}{2} \right\rceil$.

At this point, $a = b$. Set $n := a = b$.

Notice that the size of the interval $[a, b]$ is cut in (about) half at each step, so that this loops at most $O(\log_2 N)$ times, thus this algorithm makes only polynomially many calls to IFD. Also, every other computation in this algorithm (i.e., ceiling, sum, divide by 2) is polynomial time. Therefore, this entire algorithm (assuming it works) is a polynomial time reduction of IFS to IFD.

To prove it actually works, note that at the end of each loop, there is always a factor of N less than or equal to b. Notice that if at any point $a + 2 \leq b$, then

$$a < a + 1 = \left\lceil \frac{a + (a + 2)}{2} \right\rceil \leq \left\lceil \frac{a + b}{2} \right\rceil \leq \left\lceil \frac{(b - 2) + b}{2} \right\rceil = b - 1 < b.$$

Thus, the only way to get $a = b$ at the end of a loop is if at the beginning of that loop, $a + 1 = b$. Moreover, in this case, the only way to get $a = b$ is if the answer is "no". Therefore, N has a factor less than or equal to n (because $n = b$), but does not have a factor less than or equal to $n - 1$ (because $a = n - 1$ the last time through the loop and the answer was "no"). Thus n is a factor of N.

Bonus: In order for IFD to be in NP, we have to show that if IFD answers "yes", then it can provide some sort of *certificate*, which is a proof of the "yes" answer that can be checked in polynomial time. So, if IFD answers "yes" for $N > 1$ and k, it can provide a factor $2 \leq n \leq k$ of N as a certificate. This certificate can be used to prove that N has a factor less than or equal to k in polynomial time by just dividing N by n and showing that the remainder is zero. Since integer division is polynomial time, IFD is in NP. ■