Contents

2 Discrete Logarithms in Cryptography **1**

2.1 Primitive Roots and Discrete Logarithms 1

2.1.1 Primitive Roots 1

2.1.2 Discrete Logarithms 4

2.2 Diffie-Hellman Key Exchange 5

2.3 The ElGamal Encryption System 7

2.4 Computation of Discrete Logarithms 8

2.4.1 The Pohlig-Hellman Algorithm 9

2.4.2 Baby-Step Giant-Step 11

2.4.3 Pollard's ρ-Algorithm for Logarithms 12

2.4.4 Sieving Methods 13

2 Discrete Logarithms in Cryptography

In the previous chapter, we introduced public-key cryptography and discussed how to construct several public-key cryptosystems that relied on the computational difficulty of factoring large integers. In this chapter, we will introduce and study another computationally difficult number theory problem, that of computing discrete logarithms, with an eventual goal of using that problem as the basis for cryptographic protocols. We will specifically discuss the ElGamal public-key cryptosystem and the Diffie-Hellman key exchange procedure, and then give some methods for computing discrete logarithms.

2.1 Primitive Roots and Discrete Logarithms

- Recall that if u is a unit modulo m, that the order of u is the smallest positive integer k such that $u^k \equiv 1 \pmod{m}$. Earlier, we proved a few basic properties about orders:
  - If u is a unit modulo m and $u^n \equiv 1 \pmod{m}$, then the order of u divides n.
  - If u has order k modulo m, then the order of u^n modulo m is $k/\gcd(n,k)$. In particular, if n and k are relatively prime, then u^n also has order k.
  - If $u^d \equiv 1 \pmod{m}$, and $u^{d/p} \not\equiv 1 \pmod{m}$ for any prime divisor p of d, then u has order d modulo m.
  - If u has order k and w has order l, where k and l are relatively prime, then uw has order kl.

2.1.1 Primitive Roots

- Euler's Theorem says that the order of any element modulo m divides $\varphi(m)$. We might wonder: can the order actually equal $\varphi(m)$? The answer is yes, and such elements are quite useful:
- Definition: If u is a unit modulo m and the order of u is $\varphi(m)$, we say that u is a primitive root modulo m.
  - Example: The powers of 2 modulo 5 are 2, 4, 3, and 1, so 2 is a primitive root mod 5 (since it has order 4). Similarly, we can check that 3 is also a primitive root mod 5.

- Example: The powers of 2 modulo 9 are 2, 4, 8, 7, 5, and 1, so 2 is a primitive root mod 9 (since it has order $6 = \varphi(9)$).
- Non-Example: There is no primitive root modulo 15: the units are 1 (order 1), 2 (order 4), 4 (order 2), 7 (order 4), 8 (order 4), 11 (order 2), and 14 (order 2), and none of these is a primitive root.

- If there is a primitive root modulo m, then every unit can be written in terms of that primitive root (which is the reason for the term "primitive root"):
- Proposition: A unit u is a primitive root modulo m if and only if every unit modulo m is congruent to a power of u.
  - We can see this in the examples above: for example, the units modulo 5 are 1, 2, 3, and 4, and they are congruent mod 5 to 2^0, 2^1, 2^3, and 2^2 respectively.
  - Proof: If u is a primitive root modulo m, then by definition each of $u^1, u^2, \cdots, u^{\varphi(m)}$ is distinct modulo m. Since there are $\varphi(m)$ elements in this list and they are all units, this means they represent each of the invertible residue classes modulo m.
  - For the other direction, if the powers of u exhaust all of the different residue classes modulo m, then the order of u must be at least $\varphi(m)$ (since otherwise there would be fewer than $\varphi(m)$ distinct powers of u modulo m), but since the order of u divides $\varphi(m)$ by Euler's Theorem, the order must be exactly $\varphi(m)$.

- We would now like to know: when does there actually exist a primitive root modulo m? We start with primes:
- Theorem (Primitive Roots Mod p): For any prime p, there exists a primitive root modulo p.
  - The proofs of this theorem (which we will omit) are somewhat nonconstructive: they prove the second statement, and then show that a primitive root exists modulo p without actually constructing it explicitly.
  - In practice, it is not that difficult to find a primitive root (if one exists) by simply trying small values and checking whether the order is equal to $\varphi(p) = p - 1$.

- Example: Find a primitive root modulo 11.
  - We try checking whether 2 is a primitive root modulo 11.
  - The order of 2 must divide $\varphi(11) = 10$, and we see that $2^2 \not\equiv 1 \pmod{11}$ and $2^5 \not\equiv 1 \pmod{11}$, so the order divides neither 2 nor 5.
  - Therefore, the order of 2 must be 10, so $\boxed{2}$ is a primitive root modulo 11.

- If we have a primitive root modulo p, we can use it to obtain a primitive root modulo larger powers of p.
- Proposition: If a is a primitive root modulo p for p an odd prime, then a is a primitive root modulo p^2 if $a^{p-1} \not\equiv 1 \pmod{p^2}$. In the event that $a^{p-1} \equiv 1 \pmod{p^2}$, then $a + p$ is a primitive root modulo p^2. Furthermore, if b is a primitive root modulo p^2, then b is a primitive root modulo p^k for each $k \geq 3$. If b is odd then b is also primitive root modulo $2p^k$, and if b is even then $b + p^k$ is a primitive root modulo $2p^k$.
  - These results are fairly straightforward calculations using the binomial theorem. The details are not especially enlightening, so we will omit them.

- Example: Find a primitive root modulo 11^2, modulo $2 \cdot 11^2$, and modulo 11^{100}.
  - We showed above that 2 is a primitive root modulo 11.
  - We can easily compute $2^{10} = 1024 \equiv 56 \pmod{11^2}$, so the proposition above assures us that $\boxed{2}$ is also a primitive root modulo 11^2.
  - Then by the proposition, $\boxed{2}$ is also primitive root modulo 11^d for any $d \geq 2$ hence (in particular) for $d = 100$.
  - Likewise, since 2 is even, we see that $\boxed{2 + 11^2 = 123}$ is a primitive root modulo $2 \cdot 11^2$.

- In general, the above results will allow us to find primitive roots modulo p^k or $2p^k$ for any $k \geq 1$ and and odd prime p. It turns out that these are essentially all the cases in which primitive roots exist:

- Theorem (Existence of Primitive Roots): There exists a primitive root modulo m if and only if $m = 1, 2, 4$ or m is of the form p^k or $2p^k$ for an odd prime p and some $k \geq 1$.
  - The existence part follows essentially from the propositions above. The remaining part of the theorem then requires showing that there do not exist primitive roots modulo m for any other such m: the idea is simply to show that there are four elements whose square is equal to 1 modulo m. (We omit the details.)
  - In practice, it is not that difficult to find a primitive root (if one exists) by simply trying small values and checking whether the order is equal to $\varphi(m)$.
- When a primitive root exists, we can in fact say exactly how many there are:
- Proposition: If there exists a primitive root modulo m, then there are precisely $\varphi(\varphi(m))$ primitive roots modulo m.
  - Proof: Suppose that there exists a primitive root u modulo m, whose order is therefore $\varphi(m)$.
  - We know that the invertible residue classes modulo p are represented by $u^1, \ldots, u^{\varphi(m)}$, so it suffices to determine how many of these have order $\varphi(m)$.
  - Since the order of u^k is $\varphi(m)/\gcd(k, \varphi(m))$, we see that u^k is a primitive root if and only if k is relatively prime to $\varphi(m)$.
  - There are $\varphi(\varphi(m))$ such k, so there are $\varphi(\varphi(m))$ primitive roots modulo m.
- Example: Find a primitive root modulo 54 and the number of primitive roots modulo 54.
  - Since $54 = 2 \cdot 3^3$ there is a primitive root modulo 54.
  - It is easy to see that 2 is a primitive root modulo 3, and since $2^{3-1} \not\equiv 1 \pmod 9$ we see that 2 is also a primitive root modulo 9 and hence modulo 27 as well.
  - Since 2 is even, we conclude that $\boxed{29}$ is a primitive root modulo 54.
  - The number of primitive roots is $\varphi(\varphi(54)) = \varphi(18) = \boxed{6}$. (Aside from 29, the others are 5, 11, 23, 41, and 47.)
- In general, if p is prime then $\varphi(\varphi(p))$ will be roughly the same size as p, meaning that a reasonable proportion of residue classes modulo p will be units.
  - Specifically, $\varphi(\varphi(p)) = \varphi(p-1) = (p-1) \cdot \prod_{q_i | p,\, q_i \text{prime}} (1 - 1/q_i)$, and the product on the right only depends on the primes dividing $p - 1$.
  - So for example, if $p - 1$ has at most 20 distinct prime divisors (which will be the case fairly often), at least 15% of the residue classes modulo p will be primitive roots, and even if $p - 1$ has 40 distinct prime divisors, over 10% of the residue classes will necessarily be primitive roots.
- For completeness we restate an earlier result which can be used easily to check whether a given residue class is a primitive root modulo p, provided we know the factorization of $p - 1$:
- Proposition: If p is an odd prime and u is a unit with $u^{(p-1)/q} \not\equiv 1 \pmod p$ for any prime divisor q of $p - 1$, then u is a primitive root modulo p.
  - The point is that the order of u divides $p - 1$ by Fermat's little theorem, and if the order of u were smaller than $p - 1$ then $u^{(p-1)/q}$ would be congruent to 1 for at least one prime divisor q of $p - 1$.
- Example: Find a primitive root modulo $p = 2394863$, and the proportion of residue classes modulo p that are primitive roots.
  - First we factor $p - 1 = 2 \cdot 37 \cdot 32363$ using whichever factorization algorithm we prefer.
  - Now we search for residues u such that none of $u^{(p-1)/2}$, $u^{(p-1)/37}$, and $u^{(p-1)/32363}$ is congruent to 1 modulo p.
  - Here is a short table of such a check:

	$u^{(p-1)/2}$	$u^{(p-1)/37}$	$u^{(p-1)/32363}$
$u = 2$	1	14871	23729
$u = 3$	1	50374	2128656
$u = 5$	-1	2184105	929101

- Since we see that $u = 5$ has each of $u^{(p-1)/2}$, $u^{(p-1)/37}$, and $u^{(p-1)/32363}$ not congruent to 1 modulo p, we see that $\boxed{5}$ is a primitive root modulo p.
- We also compute $\varphi(\varphi(p)) = 1 \cdot 36 \cdot 32362$, so the proportion is $\varphi(\varphi(p))/p \approx 48.6\%$ of the residue classes. (From this perspective, the fact that we had to try 3 residues means we were less lucky than expected!)

2.1.2 Discrete Logarithms

- From our results, if u is a primitive root modulo m, then for any unit a there is a k such that $a \equiv u^k \bmod m$. We can generalize this idea:
- Definition: If b is a unit modulo m and a is another unit with $a \equiv b^d \pmod m$, we say that d is the discrete logarithm of a modulo m to the base b, and write $d = \log_b(a)$.
  - Note: Implicitly, we consider the discrete logarithm to be defined only modulo the order of b. Some authors instead define the discrete logarithm to be the smallest positive integer such that $a \equiv b^d \pmod m$ provided one exists, but (like with the definition of modular congruence) this definition is somewhat too restrictive.
  - The reason this map is called the discrete logarithm is because its definition is analogous to that of the usual logarithm: $\log_b(a) = d \pmod k$ is equivalent to $a \equiv b^d \pmod m$, where k is the order of b modulo m. (Compare to the definition of the real-valued logarithm: $\log_b(y) = x$ is equivalent to $y = b^x$.)
  - Example: Modulo 14, we have $\log_3 11 = 4$ since $3^4 \equiv 11 \pmod{14}$. It is better to write $\log_3 11 \equiv 4 \pmod 6$, since the order of 3 modulo 14 is 6.
- As we would expect, it is easy to see that the discrete logarithm obeys the standard rules of logarithms.
  - Specifically, suppose that k is the order of b modulo m.
  - Then $\log_b(ac) \equiv \log_b(a) + \log_b(c) \pmod k$ and $\log_b(a^r) \equiv r \log_b(a) \pmod k$ for any integer r and any residue classes a and c whose discrete logarithms to the base b are defined.
- Example: Find the discrete logarithms of each unit modulo 11 to the base 2.
  - Since 2 is a primitive root modulo 11, we can write each unit as a power of 2. The simplest way to do this is simply to compute each of the values 2^0, 2^1, ... , 2^{10} modulo 11; here is a table of the results:

n	1	2	3	4	5	6	7	8	9	10
$\log_2 n$	0	1	8	2	4	9	7	3	6	5

  - Observe, for example, that $3 \cdot 6 \equiv 7 \pmod{11}$, and $\log_2(3) + \log_2(6) \equiv \log_2(7) \pmod{10}$, since 10 is the order of 2 modulo 11.
  - Likewise, $3^3 \equiv 5 \pmod{11}$, and $3\log_2(3) \equiv \log_2(5) \pmod{10}$.
- Having a table of discrete logarithms relative to a primitive root modulo m is very useful for computations.
  - For example, it allows for very rapid multiplication and exponentiation, in the same manner as usual logarithms do. This is not terrifically helpful because there already exist fast algorithms for these procedures.
  - More usefully, having a table of discrete logarithms also allows us to compute nth roots, if they exist.
- Example: Solve the equation $x^4 \equiv 9 \pmod{11}$.
  - We could, of course, just try all of the units modulo 11 to see which ones work.

- A more efficient general method is to take discrete logarithms to the base 2: we obtain $\log_2(x^4) \equiv \log_2(9) \pmod{10}$, or $4 \log_2 x \equiv 6 \pmod{10}$.
- Since $\gcd(4, 10) = 2$ this congruence is equivalent to $2 \log_2(x) \equiv 3 \pmod{5}$, which has the solution $\log_2(x) \equiv 4 \pmod{5}$. Modulo 10 there are two solutions: 4 and 9.
- Exponentiating then yields that there are two solutions to the original congruence: $x \equiv 2^4, 2^9 \bmod 11$, or equivalently $x \equiv \boxed{5, 6} \bmod 11$.

- In general, it is believed to be difficult to evaluate discrete logarithms or to extract roots modulo m.
  - Ultimately, the difficulty of extracting roots and evaluating discrete logarithms is at the heart of the RSA cryptosystem, which relies on the difficulty of extracting an eth root modulo m.
  - Our goal in the subsequent sections will be to show how to use discrete logarithms to design cryptographic protocols.

2.2 Diffie-Hellman Key Exchange

- Public-key protocols are fast for small messages, but if Alice needs to send Bob gigabytes of encrypted data, even a very fast implementation of RSA will take an unreasonably long time to encode and decode.
  - Symmetric cryptosystems generally do not require nearly as much computation and can be done comparatively efficiently even for large amounts of data.
  - Thus, in practice, most efficient cryptographic protocols will require some sort of "key exchange", wherein Alice and Bob must somehow decide what encryption key to use for their symmetric cryptosystem.
  - One way to do this is to use an asymmetric cryptosystem to send the key: Alice chooses a key, encrypt it using Bob's public key, and send it to Bob: then Bob can decrypt the message and obtain the key.

- We will now describe a different procedure for key exchange. The general idea was initially conceived of by Ralph Merkle and a specific implementation was published by Whitfield Diffie and Ralph Hellman in 1976 (thus predating the unclassified version of RSA by about 1 year).

- Their procedure, known as Diffie-Hellman key exchange, is as follows:
  - First, Alice and Bob jointly choose a large prime number p, along with a primitive root g modulo p.
    - Finding a pair (g, p) where p is a large prime and g is a primitive root mod p is fairly straightforward.
    - For a particular choice of p it can be quite difficult to find a primitive root unless the factorization of $p - 1$ is known, so in practice one chooses p in such a way that $p - 1$ has a convenient factorization: it is then easy to test whether a given element r is a primitive root modulo p using our previous procedures.
    - One way to find such a p is to choose a random large number and then search through primes larger than that, attempting at each stage to factor $p - 1$ by removing small divisors less than some particular bound (e.g., 10^6) and then checking the remaining term using a primality test.
  - Alice chooses a secret integer a, and sends Bob the value of $g^a \pmod{p}$.
  - Bob chooses a secret integer b, and sends Alice the value of $g^b \pmod{p}$.
  - Then the secret key s is given by $g^{ab} \pmod{p}$, which both Alice and Bob can compute.
    - Alice knows a, and has the value of g^b from Bob, so she needs only raise g^b to the ath power.
    - Similarly, Bob knows b and has the value of g^a from Alice, so he needs only raise g^a to the bth power.

- If Eve is eavesdropping on the conversation, she will have the values of p, along with g, g^a, and g^b modulo p, and she wants to compute the secret key $g^{ab} \pmod{p}$.
  - In order to do this, Eve would essentially need to compute one of the exponents a and b; since g is a primitive root, this is equivalent to calculating the discrete logarithm $\log_g(g^a)$ or $\log_g(g^b)$ modulo $p - 1$.
  - Computation of discrete logarithms is very difficult, in much the way that factoring large integers is difficult. (We will postpone our discussion of this question to a later section.)

- But in general, it is expected that if p is large enough, roughly on the order of computing arbitrary discrete logarithms modulo p is computationally intractable.

- Example: Alice and Bob decide to construct a secret shared key modulo $p = 227$, using the primitive root $g = 2$.
  - Alice chooses her exponent to be $a = 44$, and computes $2^{44} \equiv 171 \pmod{p}$. She sends the value $g^a \equiv 171$ to Bob.
  - Bob chooses his exponent to be $b = 175$, and computes $2^{175} \equiv 201 \pmod{p}$. He sends the value $g^b \equiv 201$ to Alice.
  - Alice now computes the shared key $g^{ab} \equiv 201^{44} \pmod{p}$ and obtains the value $s \equiv 160 \pmod{p}$.
  - Bob also computes the shared key $g^{ab} \equiv 171^{175} \pmod{p}$ and obtains the same value $s \equiv 160 \pmod{p}$.
  - Alice and Bob now can send each other messages encrypted (in whatever way they want) using their shared secret key $s = 160$.

- We will also observe that Diffie-Hellman key exchange can be generalized to any number of participants.
  - For example, if Alice, Bob, and Carol wish to construct a secret shared key together, they collectively agree on a fixed prime p and primitive root g modulo p, and choose their own encryption exponents a, b, and c.
  - Alice publishes g^a, Bob publishes g^b, and Carol publishes g^c (everything modulo p).
  - Next, Alice uses g^b and g^c to compute g^{ab} and g^{ac} mod p, and publishes these values.
  - Bob uses g^c to compute g^{bc} mod p, and publishes it.
  - The secret key is then $s = g^{abc}$: Alice uses g^{bc} to compute s, Bob uses g^{ac} to compute s, and Carol uses g^{ab} to compute s.
  - If Eve is eavesdropping on the conversation, she has the values of g, g^a, g^b, g^c, g^{ab}, g^{ac}, and g^{bc}. However, in order for her to compute g^{abc}, she would need to compute the discrete logarithm of g^a, g^b, or g^c, which is still just as difficult as it was before.

- Here are a few basic attacks on Diffie-Hellman:

- Attack 1 (Discrete logarithm computation): If Eve wants to determine g^{ab} given g, g^a, g^b, she could simply evaluate the discrete logarithm $\log_g(g^a)$ to compute a, and then evaluate $(g^b)^a$.
  - Again, like with factorization, it is believed that general discrete logarithm computation is difficult with a standard (i.e., non-quantum) computer, provided the prime p is sufficiently large and not of any particularly special form (e.g., not such that $p - 1$ only has small prime divisors).

- Attack 2 (Man-in-the-middle): A malicious individual, Mallory, could intercept the original key exchanges and conduct separate key-exchanges with Alice (with Mallory pretending to be Bob) and Bob (with Mallory pretending to be Alice).
  - Then, Mallory will be able to decode messages sent from Alice, and then re-encrypt them to send to Bob. As far as Alice and Bob can tell, they are communicating with each other, since their messages are received correctly, at least as long as Mallory is in the middle decoding and re-encoding the messages.
  - If at any point Mallory fails to do this (and a message is transmitted directly from Alice to Bob), Alice and Bob's keys will not agree and they will realize that their communications have been intercepted, since they will no longer be able to decode each other's messages.
  - The problem is that the algorithm above does not authenticate Alice and Bob to one another before creating the key. There are various modifications to the basic Diffie-Hellman algorithm that allow for mutual authentication: they are sufficiently distinct from the basic algorithm that we will postpone their discussion until later.

2.3 The ElGamal Encryption System

- The RSA public-key cryptosystem ultimately relies on the difficulty of integer factorization. We will now describe the ElGamal public-key cryptosystem, first described by Taher Elgamal in 1985, whose security relies on the difficulty of computing discrete logarithms.
- First, Bob must create his public key.
  - To do this, he chooses a prime p and a primitive root a modulo p such that it is difficult to compute discrete logarithms modulo p. (Typically this is done by ensuring that $p-1$ has a large prime divisor.)
  - Bob also chooses an integer d with $0 < d < p-1$, and computes $b = a^d \pmod{p}$.
  - Bob then publishes the three values (p, a, b), which serve as his public key.
- Now suppose that Alice wants to send Bob a message.
  - Alice converts her message into an integer m modulo p in some agreed-upon manner.
  - Alice then chooses a random integer k with $0 < k < p-1$ and computes $r = a^k \pmod{p}$ and $t = b^k m \pmod{p}$.
  - Finally, she sends the pair (r, t) to Bob.
- If Bob has received a ciphertext pair (r, t), he wishes to recover the value of m.
  - To do this, Bob simply computes $t \cdot r^{-d} \equiv (b^k m)(a^{-kd}) \equiv (a^{kd} m)(a^{-kd}) \equiv m \pmod{p}$.
- Example: If Bob uses ElGamal with $p = 44927$, $a = 7$, $d = 22105$, find Bob's public key, encode the message $m = 10101$, and then decode the associated ciphertext.
  - First, we compute Bob's public key: we have $b = a^d \equiv 40909 \pmod{p}$, so Bob's public key is $(p, a, b) = \boxed{(44927, 7, 40909)}$.
  - To encode, we choose a random k with $0 < k < p-1$: let us take $k = 6708$. We then compute $r = a^k \equiv 12510 \pmod{p}$ and $t = b^k m \equiv 12749 \pmod{p}$, so the ciphertext Alice sends Bob is $\boxed{(r, t) = (12510, 12749)}$.
  - To decrypt, Bob computes $r^{-d} \equiv 11355 \pmod{p}$ and multiplies it by t to obtain the result $\boxed{m = 10101}$, as he should.
- Like with RSA, the only steps required to implement ElGamal are modular exponentiation and inversion (to compute r^{-d}) which are both very fast, but it is less obvious why the procedure is secure.
  - Suppose Eve intercepts the transmitted information: she will obtain p, a, b, r, and t, and she wants to compute $m = t \cdot b^{-k} \equiv t \cdot a^{-dk} \equiv t \cdot r^{-d}$ modulo p.
  - If Eve knows d then she can decrypt using the same procedure Bob uses. However, in order to find d from Bob's public key, Eve would need to compute the discrete logarithm $\log_a b$, which we assume she cannot do.
  - Furthermore, since Alice chooses k randomly, $r = a^k$ will be a random integer modulo p, as will $t = b^k m$ (since b^k is likewise random) provided $m \neq 0$.
  - Knowing r alone does not help, because in order to compute k Eve would need to evaluate the discrete logarithm $\log_a r$. Knowing t does not help much either, because in order for Eve to compute m she would have to know the value of b^k, which in turn would require knowing the value of k.
- In order to compute any one of the desired quantities to decrypt an ElGamal ciphertext, it seems that Eve would essentially have to evaluate a discrete logarithm.
  - This is not a proof, of course, and it is not actually known whether breaking ElGamal encryption is equivalent to evaluating discrete logarithms.

- However, it can be shown that breaking ElGamal encryption and breaking Diffie-Hellman are equivalent to one another, in the sense that an algorithm for decrypting ElGamal ciphertexts modulo p can be used to break Diffie-Hellman mod p, and vice versa:

- Proposition: The problems of decrypting arbitrary ElGamal ciphertexts modulo p and breaking arbitrary Diffie-Hellman procedures mod p are equivalent.

  - Proof: Suppose first that we have an algorithm that can decrypt an arbitrary ElGamal ciphertext (r, t) with associated public key (p, a, b) to produce the message $m \equiv t \cdot r^{-\log_a b} \pmod{p}$, and we wish to break a Diffie-Hellman problem of computing the value $g^{xy} \pmod{p}$ given the values (p, g, c_x, c_y) where $c_x = g^x \pmod{p}$, $c_y = g^y \pmod{p}$.
  - To do this, give the ElGamal algorithm the data (p, a, b, r, t) with $a = g$, $b = c_x$, $t = 1$, and $r = c_y$: it will output the message $m = 1 \cdot (g^y)^{-\log_g(g^x)} \equiv g^{-xy} \pmod{p}$. Then $g^{xy} \equiv m^{-1} \pmod{p}$ can be computed immediately.
  - Conversely, suppose we have an algorithm that can break an arbitrary Diffie-Hellman problem of computing the value $g^{xy} \pmod{p}$ given the values (p, g, c_x, c_y) where $c_x = g^x \pmod{p}$, $c_y = g^y \pmod{p}$, and we wish to decrypt an arbitrary ElGamal ciphertext (r, t) with associated public key (p, a, b) to produce the message $m \equiv t \cdot r^{-\log_a b} \pmod{p}$.
  - To do this, give the Diffie-Hellman algorithm the data (p, g, c_x, c_y) where $g = a$, $c_x = b$, and $c_y = r$: it will then output the value $c_x^{\log_g c_y} = b^{\log_a r} = b^k$. We can then compute $m \equiv t \cdot b^{-k} \pmod{p}$ immediately.

- We will also remark that, unlike the basic version of RSA where it is easy to verify that a given ciphertext c has a claimed decryption m (we simply compute $m^e \pmod{N}$ and check whether it is equal to c), it is not so easy to determine whether a claimed ciphertext (r, t) for ElGamal encryption actually corresponds to a particular decrypted message m.

  - In fact, this problem is equivalent to the "decision Diffie-Hellman problem": that of determining whether a given set of numbers (p, g, c_x, c_y, c_{xy}) has the property that $c_x = g^x \pmod{p}$, $c_y = g^y \pmod{p}$, and $c_{xy} = g^{xy} \pmod{p}$ for some values of x and y.
  - The proof (which we will omit) is essentially the same as the one we gave above.

- There are several attacks on ElGamal encryption similar to those for RSA; we will mention a few basic ones:

- Attack 1 (Discrete logarithm computation): If Eve wants to determine m, she can first compute Alice's value of k by evaluating the discrete logarithm $k = \log_a r$, and then compute $m \equiv t \cdot b^{-k} \pmod{p}$, or she could compute Bob's value of d by evaluating $d = \log_a b$ and then compute m the same way Bob does.

  - Again, like with factorization, it is believed that general discrete logarithm computation is difficult with a standard (i.e., non-quantum) computer, provided the prime p is sufficiently large and not of any particularly special form (e.g., not such that $p - 1$ only has small prime divisors).

- Attack 2 (Duplicate k, partial known plaintext): Suppose Alice sends two messages m_1 and m_2 to Bob and reuses the same value of k for each message.

  - Eve would then intercept the two pairs (r_1, t_1) and (r_2, t_2) where $t_1 = b^k m_1 \pmod{p}$ and $t_2 = b^k m_2 \pmod{p}$.
  - If Eve also knows the plaintext m_1, she can easily compute the plaintext m_2, since

$$t_2 t_1^{-1} m_1 = (b^k m_2)(b^k m_1)^{-1} m_1 \equiv m_2 \pmod{p}.$$

2.4 Computation of Discrete Logarithms

- In this section we give a few methods for computing discrete logarithms. In general, discrete logarithm computation appears to be of approximately comparable difficulty to factoring, and many of the algorithms share the same kinds of underlying ideas.

- If we can solve discrete logarithm problems where the base is a given primitive root a, then the change-of-base formula gives us a simple linear congruence $\log_a x \cdot \log_x y \equiv \log_a y \pmod{p-1}$ which we can immediately solve for $\log_x y$.
- Thus, we will assume throughout that we are seeking to compute a discrete logarithm whose base is a primitive root, since that is the most general case.
- We will adopt the specific notation of solving the congruence $a^d \equiv b \pmod{p}$ where a is a primitive root modulo the prime p and (a, b, p) are given.

2.4.1 The Pohlig-Hellman Algorithm

- Our first method for computing discrete logarithms is based off a principle similar to Pollard's $(p-1)$-algorithm.
- Explicitly, suppose a and b are units modulo an odd prime p: then the value of the discrete logarithm $d = \log_a b$ is a residue class modulo $p-1$.
  - By the Chinese Remainder Theorem, it is therefore sufficient to compute the value of d modulo each prime-power divisor of $p-1$. In other words, if we have the prime factorization $p-1 = \prod_i q_i^{c_i}$, we want to find d modulo $q_i^{c_i}$ for each i. For ease of notation we will now drop the subscript i.
  - Suppose the base-q expansion of d is $d = d_0 + d_1 q + d_2 q^2 + \cdots + d_{c-1} q^{c-1} \pmod{q^c}$, with each digit d_j satisfying $0 \le d_i \le q-1$: we then want to find each of the digits d_j.
  - The key observation is that $\left(\frac{p-1}{q}\right)(d - d_0) = (p-1)(d_1 + d_2 q + \cdots + d_{c-1} q^{c-2})$ is a multiple of $p-1$, meaning that $\left(\frac{p-1}{q}\right) d$ and $\left(\frac{p-1}{q}\right) d_0$ are congruent modulo $p-1$.
  - Thus, by Fermat's little theorem, we see that $a^{d_0(p-1)/q} \equiv a^{d(p-1)/q} \equiv b^{(p-1)/q} \pmod{p}$.
  - Therefore, the digit d_0 will be the unique value between 0 and $q-1$ inclusive such that $a^{d_0(p-1)/q} \equiv b^{(p-1)/q} \pmod{p}$.
  - To compute additional digits we can use the same idea: for d_1, we can pull off the first two terms to write $\left(\frac{p-1}{q^2}\right)(d - d_0 - d_1 q) = (p-1)(d_2 + d_3 q + \cdots)$ so that $\left(\frac{p-1}{q^2}\right)(d - d_0)$ and $\left(\frac{p-1}{q}\right) d_1$ are congruent modulo $p-1$.
  - Then by Fermat's little theorem, $a^{d_1(p-1)/q} \equiv a^{(d-d_0)(p-1)/q^2} \equiv b^{(p-1)/q^2} a^{-d_0(p-1)/q^2} \pmod{p}$.
  - Thus, the digit d_1 will be the unique value between 0 and $q-1$ inclusive such that $a^{d_1(p-1)/q} \equiv b^{(p-1)/q^2} a^{-d_0(p-1)/q^2}$.
  - We can continue iterating this procedure to compute all of the remaining digits.
- Here is a more algorithmic procedure for computing the discrete logarithm $d = \log_a b$ modulo p, where p is a prime and a is a primitive root. (In fact a does not actually need to be a primitive root, although in most cases that is what we are most interested in.)
- Algorithm (Pohlig-Hellman): Suppose that q^c is a prime power dividing $p-1$. To compute d modulo q^c, compute the values $a^{k(p-1)/q}$ for each $0 \le k \le q-1$. Set $b_0 = b$ and then take d_0 to be the value for which $a^{d_0(p-1)/q} \equiv b_0^{(p-1)/q} \pmod{p}$. Now for each $1 \le i \le c-1$, set $b_i = b_{i-1} \cdot a^{-d_{i-1} q^{i-1}}$ and take d_i to be the value for which $a^{d_i(p-1)/q} \equiv b_i^{(p-1)/q^{i+1}} \pmod{p}$. Then the value d is then equal to $d_0 + d_1 q + d_2 q^2 + \cdots + d_{c-1} q^{c-1}$. Finally, to compute the discrete logarithm $d = \log_a b$, assemble all of the individual values d modulo q^c via the Chinese remainder theorem to obtain the value of d modulo $p-1$.
  - As with Pollard's $(p-1)$-algorithm, the idea behind the Pohlig-Hellman algorithm is that if $p-1$ only has small prime divisors, then we can evaluate all of the ingredients in the computation rapidly.
  - Explicitly, to compute d modulo q^c, we need to evaluate the q values $a^{k(p-1)/q}$ for each $0 \le k \le q-1$, and then match c values from this list. Therefore, the number of steps required to implement the Pohlig-Hellman algorithm to compute a discrete logarithm modulo p is roughly equal to the largest prime divisor q of $p-1$.

- Example: Use the Pohlig-Hellman algorithm to find the discrete logarithm of $b = 11850$ to the base $a = 5$ modulo $p = 24697$.
  - First, we factor $p - 1 = 2^3 3^2 7^3$. (From here it is also easy to verify that $a = 5$ is a primitive root modulo p.)
  - We need to compute the discrete logarithm d modulo 2^3, 3^2, and 7^3.
  - For $q^c = 2^3$:
    - First, we we pre-evaluate the quantities $a^{k(p-1)/q}$ for $k = 0, 1$, yielding the values 1 and $-1 \pmod p$.
    - Next we compute the required quantities b_i and d_i for $0 \le i \le 2$: if $b_i^{(p-1)/q^{i+1}}$ is 1 then $d_i = 0$, and if $b_i^{(p-1)/q^{i+1}}$ is -1 then $d_i = 1$.

i	0	1	2
b_i	11850	11850	474
$b_i^{(p-1)/q^{i+1}}$	1	-1	1
d_i	0	1	0

    - Then $d = 0 + 1 \cdot 2 + 0 \cdot 2^2 = 2$ modulo 8.
  - For $q^c = 3^2$:
    - First, we we pre-evaluate the quantities $a^{k(p-1)/q}$ for $k = 0, 1, 2$, yielding the values 1, 3067, 21629 $\pmod p$.
    - Next we compute the required quantities b_i and d_i for $0 \le i \le 3$: if $b_i^{(p-1)/q^{i+1}}$ is 1 then $d_i = 0$, if it is 3067 then $d_i = 1$, and if it is 21629 then $d_i = 2$.

i	0	1
b_i	11850	2370
$b_i^{(p-1)/q^{i+1}}$	3067	3067
d_i	1	1

    - Then $d = 1 + 1 \cdot 3 = 4$ modulo 9.
  - For $q^c = 7^3$:
    - First, we we pre-evaluate the quantities $a^{k(p-1)/q}$ for $k = 0, 1, \ldots, 6$:

k	0	1	2	3	4	5	6
$a^{k(p-1)/q}$	1	8955	866	172	9046	770	4887

    - Next we compute the required quantities b_i and d_i for $0 \le i \le 3$ using the table above:

i	0	1	2
b_i	11850	7922	14275
$b_i^{(p-1)/q^{i+1}}$	9046	9046	172
d_i	4	4	3

    - Then $d = 4 + 4 \cdot 7 + 3 \cdot 7^2 = 179$ modulo 7^3.
  - Finally, using the Chinese remainder theorem we can solve the simultaneous congruences $d \equiv 2 \pmod 8$, $d \equiv 4 \pmod 9$, and $d \equiv 179 \pmod{7^3}$ to obtain the solution $d \equiv \boxed{14242} \pmod{24696}$.
  - We can indeed see that this answer is correct by evaluating $5^{14242} \equiv 11850 \pmod p$ using successive squaring.

- We will remark that the speed of the Pohlig-Hellman algorithm depends on the size of the largest prime divisor of $p - 1$, which can vary quite substantially even for primes p of approximately the same size.
  - When generating a prime for use in Diffie-Hellman key exchange (or the ElGamal cryptosystem) it is comparatively unlikely to choose one such that $p - 1$ has only small prime divisors by accident.
  - For actual implementation of Diffie-Hellman, one requires a primitive root g modulo p: verifying that a particular value g is actually a primitive root requires knowing the factorization of $p - 1$.
  - Thus, for effective use of Diffie-Hellman, one should choose p such that $p - 1$ is easily factored but also has a large prime divisor.

- If, for example, one wants to generate a 250-digit prime p such that $p-1$ has a large prime divisor, one could first generate a 200-digit prime p_0 and a random 50-digit number k, and then test the numbers $p = (k+r)p_0 + 1$ for integers $r \geq 0$ until a prime is found. By construction, $p-1$ will then have the 200-digit prime p_0 as a divisor. The remaining divisor of $p-1$ is small enough that it can be factored directly.

2.4.2 Baby-Step Giant-Step

- Another way we can try to compute discrete logarithms is by using a "meet in the middle" procedure that exploits an idea similar to the birthday paradox used by Pollard's ρ-algorithm.
- Algorithm (Baby-Step Giant-Step): In order to find a solution to $a^d \equiv b \pmod{p}$, choose an integer N such that $N^2 \geq p$. Then compute two lists: the values $a^x \pmod{p}$ for all $0 \leq x \leq N-1$ and the values $ba^{-Ny} \pmod{p}$ for all $0 \leq y \leq N-1$. Then compare the two lists to find an element that is on both lists: then if $a^x \equiv b\,a^{-Ny} \pmod{p}$, we then obtain a solution $d = x + Ny$.
  - The reason for the name of the algorithm is that the computations for the first list are the "baby steps" (each step increases the exponent of a by 1) while the computations for the second list are the "giant steps" (each step decreases the exponent of a by N).
  - All that is necessary to explain is why we should expect the two lists to have an element in common.
  - The reason is simple: if we write d in base N, then since $N^2 \geq p > d$ we know that d has at most two digits, so $d = d_0 + d_1 N$ for digits $0 \leq d_0, d_1 \leq N-1$.
  - Then we will obtain the required collision when $x = d_0$ and $y = d_1$, and each of these values will be in the range we test.
  - Even if we chose the terms in the two lists randomly, we should expect to obtain a collision because we are looking for a collision modulo p among roughly $2\sqrt{p}$ elements, so by our analysis of the birthday paradox we should expect to see a collision with reasonably high probability.
  - The algorithm will require computing two lists each having $N \approx \sqrt{p}$ elements, and then comparing them. (It is not necessary to store the elements in the second list, since we only need to compare each individual element on the second list to the first list.) Since modular exponentiation / multiplication is fast, the total time and memory required are both roughly $\sqrt{p}$.
  - We will note that this algorithm, unlike many of the others we have discussed, is completely deterministic: it is guaranteed to return a result after its computation finishes, and we know exactly how much time and memory are required.
- Illustrating the method with a prime large enough to make direct computation difficult is not so easy to do, so rather than giving a realistic example, we will instead use a small prime that allows us to display all of the required tables.
- Example: Use the baby-step giant-step algorithm to find the discrete logarithm of $b = 188$ to the base $a = 3$ modulo $p = 223$.
  - Since $\sqrt{p} \approx 14.93$ we can take $N = 15$. Here is a table of the two lists we must compare:

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a^x	1	3	9	27	81	20	60	180	94	59	177	85	32	96	65
$b\,a^{-Nx}$	188	57	213	80	29	214	72	93	148	154	106	44	94	140	218

  - We see that $a^8 \equiv 94 \equiv b \cdot a^{-15(12)} \pmod{p}$, from which we obtain $a^{8+15(12)} \equiv b \pmod{p}$. Thus, the discrete logarithm is $\boxed{188}$. (The fact that it is also equal to b is merely an odd coincidence.)

2.4.3 Pollard's ρ-Algorithm for Logarithms

- Much as the baby-step giant-step algorithm is similar to Pollard's $(p-1)$-algorithm, there is also an analogue of Pollard's ρ-algorithm for computing discrete logarithms. However, it is more complicated, and does not give an appreciably faster computation procedure. Here is a brief outline of the ideas:
  - If we wish to find d for which $a^d \equiv b \pmod{p}$, it is essentially sufficient to find (s,t) and (x,y) such that $a^s b^t \equiv a^x b^y \pmod{p}$: then by taking the discrete log and rearranging we obtain the congruence $(x-s) \equiv d(y-t) \pmod{p-1}$ which we can solve for d.
  - If $y-t$ happens to have a common divisor with $p-1$ we will not obtain a unique solution for d, but if we find (s,t) and (x,y) "randomly", it is unlikely that $\gcd(y-t, p-1)$ will be very large, and we can then search through the small number of possible values of d directly to determine the correct one.
  - The idea is then to generate a list of "random" values (s_i, t_i) and search for a collision $a^{s_i} b^{t_i} \equiv a^{s_j} b^{t_j} \pmod{p}$.
  - If we simply do this directly, then by our analysis of the birthday paradox we would need approximately $2\sqrt{p}$ pairs to be reasonably likely of obtaining a collision, but we would need to do roughly $2p$ comparisons to find that match, which is less efficient than simply trying all the residue classes directly.
  - Like with the ρ-algorithm for factorization, Pollard's observation is that we can speed up this process by generating the residue classes by iterating a "random function" and then searching for a repeated value that occurs in a cycle.
- Algorithm (Pollard's ρ-Algorithm for Logarithms): Divide the residues modulo p into three subsets S_a, S_b, S_s in some manner, and define the three functions $f(x) = \begin{cases} ax & \text{for } x \text{ in } S_a \\ bx & \text{for } x \text{ in } S_b \\ x^2 & \text{for } x \text{ in } S_s \end{cases}$, $g_a(x,n) = \begin{cases} n+1 & \text{for } x \text{ in } S_a \\ n & \text{for } x \text{ in } S_b \\ 2n & \text{for } x \text{ in } S_s \end{cases}$, $g_b(x,n) = \begin{cases} n & \text{for } x \text{ in } S_a \\ n+1 & \text{for } x \text{ in } S_b \\ 2n & \text{for } x \text{ in } S_s \end{cases}$. Choose random starting values a_0 and b_0, set $x_0 = a^{a_0} b^{b_0} \pmod{p}$, and then for each $i \geq 1$, set $x_i = f(x_{i-1}) \pmod{p}$, $a_i = g_a(x_{i-1}, a_{i-1}) \pmod{p-1}$, and $b_i = g_b(x_{i-1}, b_{i-1}) \pmod{p-1}$. Compare the values of the two sequences $\{x_i\}$ and $\{x_{2i}\}$: if $x_i = x_{2i}$ for some $i > 0$, then the discrete logarithm $d = \log_a b$ satisfies the congruence $(a_{2i} - a_i) \equiv d(b_{2i} - b_i) \pmod{p-1}$. If no such match is found, start over with a different value for x_0 or a different partition S_a, S_b, S_s.
  - There is a lot to unpack in this description, but the the idea is that f is a "random function" on the nonzero residue classes mod p, while g_a keeps track of the exponent of a and g_b keeps track of the exponent of b. In the set S_a, the exponent of a is increased by 1, in S_b the exponent of b is increased by 1, and in S_s both exponents are doubled. (On the exponents themselves, the maps g_a and g_b are generally unpredictable.)
  - Absent any reason to expect otherwise, we would guess that the values x_i should be essentially random after the first few iterations. Our analysis of the birthday paradox then suggests we should expect to see a match after roughly $2\sqrt{p}$ iterations.
  - In the same way as for Pollard's ρ-algorithm for factorization, the advantage lies in the fact that if $x_i \equiv x_j \pmod{p}$, then $x_{i+1} \equiv x_{j+1} \pmod{p}$: so if $t \geq i$ is any multiple of the period $j-i$, then $x_t \equiv x_{2t} \pmod{p}$. This means we can detect the periodicity of this sequence by looking only at pairs of the form (x_t, x_{2t}), which is a vast improvement over having to search all pairs (x_i, x_j).
  - Like with the factorization algorithm, we can reduce the memory requirements by defining a new sequence $y_i = x_{2i}$, where we iterate twice at each step rather than once, and then compare the values x_i and y_i at each stage.
  - Implementing the algorithm in this manner requires only storing six values (the values x_i, a_i, b_i and x_{2i}, a_{2i}, b_{2i}) and has time complexity approximately equal to $\sqrt{p}$: roughly the same amount of time as the baby-step giant-step algorithm, though far less memory.

2.4.4 Sieving Methods

- There is also a procedure analogous to the quadratic sieve for computing discrete logarithms, known as the index calculus algorithm. Here is an outline:
  - First, we choose a bound B. We then compute a^k modulo p for many values of k, and then attempt to factor the result and write it as a product of primes less than B. If this is not possible, we discard the result.
  - If we can write $a^k \equiv \prod p_i^{r_i} \pmod{p}$, then taking the discrete log of both sides gives a relation $k \equiv \sum r_i \log_a(p_i) \pmod{p-1}$ in the discrete logarithms $\log_a(p_i)$ for the primes p_i less than B.
  - If we obtain enough such relations, we can then solve the resulting system of linear congruences to find $\log_a(p_i)$ for each prime p_i less than B.
  - There are simple linear-algebra procedures for doing this by row-reducing an appropriate matrix (which is quite computationally efficient), although it is complicated slightly by the fact that the modulus $p-1$ is not prime.
  - Once we have computed the discrete logarithms $\log_a(p_i)$ for each prime p_i less than B, we then try to use the results to determine the desired discrete logarithm $\log_a(b)$.
  - To do this, we compute $b \cdot a^k$ modulo p for many values of k and attempt to write it as a product of primes less than B. If we find such a relation $b \cdot a^k \equiv \prod p_i^{s_i} \pmod{p}$, we can then compute $\log_a b = -k + \sum s_i \log_a(p_i) \pmod{p-1}$.
- Notice also that the evaluations of $\log_a(p_i)$ can be reused in subsequent discrete logarithm computations modulo p with the same base a (and using the discrete version of the change-of-base formula, for any other base).
  - The choice of the bound B will determine how much of the computation is done "ahead of time" and how much must be repeated to evaluate a particular discrete logarithm.
  - If B is large, then the initial step of the computation will take a long time, but the second stage will be very fast, since it is likely that few evaluations $b \cdot a^k$ modulo p will be required.
  - If B is small, the initial step of the computation will be shorter (though perhaps not tremendously shorter, since factorizations involving primes less than B will be correspondingly rarer as well), and the second step will be longer.
  - In practice, this sieving method is effective for primes p that are several hundred bits in length. Of course, if a particular prime p is commonly used as a discrete logarithm modulus, it could potentially become worthwhile to do the very lengthy first-stage computation with a large value of B, since the results could then be reused many times to compute many discrete logarithms.
- Like with integer factorization, there is a refinement of the quadratic sieve (the general number field sieve) that can be adapted to compute discrete logarithms. (We will omit the details.)
- The sieving algorithms run in speed that is asymptotically far faster than other methods for sufficiently large p.
  - Specifically, the computational complexity for the index calculus algorithm is approximately $e^{(2\ln p)^{1/2}(\ln\ln p)^{1/2}}$.
  - For large p, this is much smaller than the complexity $p^{1/2}$ of the baby-step giant-step algorithm.
  - Like with integer factorization, there are algorithms that could run on a quantum computer that can compute discrete logarithms much more quickly, in time polynomial in $\ln p$.

Well, you're at the end of my handout. Hope it was helpful.