6.875 Number Theory Lecture Notes

Vinod Vaikuntanathan

October 19, 2021

This evolving set of notes will serve as lecture notes for the number-theoretic portion of 6.875.

1 Groups

An Abelian group $G = (S, \circ)$ is a set S together with a binary operation $\circ : S \times S \to S$ which satisfies

- **Identity:** There is an element $\mathsf{Id} \in S$ such that for all $a \in S$, $a \circ \mathsf{Id} = \mathsf{Id} \circ a = a$.
- **Inverse:** For every $a \in S$, there is an element $b \in S$ such that $a \circ b = b \circ a = \mathsf{Id}$.
- **Associativity:** For every $a, b, c \in S$, $a \circ (b \circ c) = (a \circ b) \circ c$.
- **Commutativity:** For every $a, b \in S$, $a \circ b = b \circ a$.

Notice that we defined Abelian (or commutative) groups by default. Those will be the only type of groups that we see in this class. The definition of general groups is the same as the above, except without the commutativity property. Also, most of the time in this class, the groups we deal with will be finite, namely S will be a finite set. An occasional exception is the additive group $\mathbb{Z}$ of all integers.

Some notations.

- We will sometimes abuse notation and say that $g \in G$ when we really mean that $g \in S$.
- We will denote the multiplicative shorthand for iterated group operations by default. That is, for $g \in S$, $g^2 = g \circ g$, $g^3 = g \circ g \circ g$, and so forth.
- $g^0 = \mathsf{Id}$ and g^{-1} denotes the inverse of g.

An Example: The Additive Group $\mathbb{Z}_N$. A simple example is the additive group $\mathbb{Z}_N = (S = \{0, 1, 2, \ldots, N-1\}, +)$ consisting of integers from 0 to $N-1$, where the group operation is addition modulo N. The identity element is 0, its inverse is 0, and the inverse of $x \neq 0$ is $N - x$. We will call them the additive identity and the additive inverse, to distinguish them from their multiplicative friends who will show up soon, and who will turn out to be more useful.

Order of a Group and the order of an element. The order of a group is the number of elements in it, namely $|S|$. The order of an element $g \in S$ is the number of times one has to perform the group operation on g to get to the identity element Id. That is,

$$\mathsf{ord}(g) = \min_{i>0}\{g^i = \mathsf{Id}\} \ .$$

For example, the order of the group $\mathbb{Z}_N$ is N.

Theorem 1.1 (Lagrange's Theorem)**.** *The order of any element divides the order of the group.*

Proof. Let $g \in G$ be some group element. Note that multiplication by g defines a bijection on G. That is, $f : G \to G$ defined by $f(x) = gx$ is a one-to-one and onto function.

We first claim that $g^{|G|} = \mathsf{Id}$. Let's multiply all the group elements in two ways. Notice that on the left and the right, we are computing the product over the same set of elements since multiplying by g permutes the elements of the group. So,

$$\prod_{h\in G} h = \prod_{h\in G} (hg) = g^{|G|} \circ \Big(\prod_{h\in G} h\Big)$$

Dividing by $\prod_{h\in G} h$ from the left and the right gives us $g^{|G|} = \mathsf{Id}$.

Now let x be the order of g. Let $x' = \mathsf{gcd}(x, |G|)$. By Extended Euclid, there are integers a and b such that

$$ax + b|G| = x'$$

Therefore,

$$g^{x'} = g^{ax+b|G|} = (g^x)^a (g^{|G|})^b = (\mathsf{Id})^a (\mathsf{Id})^b = \mathsf{Id}$$

If $x' < x$, this contradicts the assumption that x' was the order of G, and therefore the smallest positive power of G that gives Id. So, it must be the case that $x' = x$ and thus $x = \mathsf{gcd}(x, |G|)$ and consequently, x divides G. □

Generator of a Group. A generator of a group G is an element of order $|G|$. In other words,

$$G = \{g, g^2, \ldots, g^{|G|} = \mathsf{Id}\}$$

For example, 1 is a generator of $\mathbb{Z}_N$ for any N.

Cyclic group. A group G is called cyclic if it has a generator. By the above, $\mathbb{Z}_N$ is always cyclic. We also know:

Theorem 1.2. *Every group whose order is a prime number, is cyclic. Moreover, every element other than the identity is a generator.*

Proof. Exercise. □

Discrete Logarithms. Let G be a cyclic group. We know that g has a generator, and that every $h \in G$ can be written as $h = g^x$ for *a unique* $x \in \{1, 2, \ldots, |G|\}$. We write

$$x = \mathsf{dlog}_g(h)$$

to denote the fact that x is the discrete logarithm of h to the base g.

We will look for groups where computing the group operations is easy (namely, polynomial time) but computing discrete logarithms is hard (namely, exponential or sub-exponential time). Our source for such groups will come from number theory, so let us move on to reviewing a few basic notions from (computational) number theory.

Discrete logarithms in $\mathbb{Z}_N$ are, for better or worse, easy. Indeed, the generators of $\mathbb{Z}_N$ are precisely those g whose greatest common divisor with N is 1. If you are told that

$$x \cdot g = h \pmod{N}$$

then x can be computed as $h \cdot g^{-1} \pmod{N}$ where g^{-1} now denotes the *modular multiplicative inverse* of g which can be found easily using the extended Euclidean algorithm. (See next section). It is at this point that we abandon $\mathbb{Z}_N$ and move on to other groups, notably the multiplicative group $\mathbb{Z}_N^*$.

2 Baby (Computational) Number Theory

Greatest Common Divisors, Euclid and Extended Euclid. The greatest common divisor (gcd) of positive integers a and b is the largest positive integer d that divides both a and b. a and b are relatively prime if their gcd is 1.

One can find d using Euclid's algorithm in time $O(n^2)$ where n is the maximum of the bit-lengths of a and b. One can do more: it turns out that there are always integers (not necessarily positive) x and y such that

$$ax + by = d$$

These integers can be found in essentially the same complexity using a modification of Euclid's algorithm, referred to as extended Euclid. We will assume that you have seen these algorithms and facts (say, in a previous course such as 6.042 or 6.006.)

Modular Arithmetic. We expect you to be familiar with modular arithmetic. For integers a and b, $a \pmod{N}$ will denote the remainder upon dividing a by N.

b is the (multiplicative) inverse of a if $ab = 1 \pmod{N}$. In this case, we say $b = a^{-1} \pmod{N}$. The multiplicative inverse of a exists if and only if $\gcd(a, N) = 1$.

Exercise: Show a polynomial-time algorithm to compute the multiplicative inverse of a given number a modulo a given N.

Modular Exponentiation. This is the operation of computing $a^b \pmod{N}$. Computing a^b over the integers and then reducing mod N is a horribly inefficient way to do this (think about it). Instead, one uses the repeated squaring algorithm that runs in time $O(n^3)$.

Operation	Time Complexity	Remarks
$a+b$	$O(n)$	grade-school addition
ab	$O(n^2)$	grade-school multiplication
	$O(n \log n)$	Harvey and van der Hoeven 2020
$\gcd(a,b)$	$O(n^2)$	Euclidean algorithm or binary gcd algorithm
$a \pmod N$	$O(n^2)$	
$a+b \pmod N$	$O(n^2)$	
$ab \pmod N$	$O(n^2)$	
$a^{-1} \pmod N$	$O(n^2)$	extended Euclidean algorithm
$a^b \pmod N$	$O(n^3)$	repeated squaring algorithm
Checking if a given p is prime	$O(n^2 \log 1/\epsilon)$ rand.	Miller-Rabin
	$O(n^6)$ det.	Agrawal-Kayal-Saxena 2002 and followups
Factoring N	$2^{O(n^{1/3}(\log n)^{2/3})}$	Number field sieve
Discrete log in $\mathbb{Z}_N^*$	$2^{O(n^{1/3}(\log n)^{2/3})}$	Number field sieve
Discrete log in any group G	$O(\sqrt{\lvert G \rvert})$	baby step-giant step algorithm

Figure 1: The complexity of basic operations with numbers. n denotes the input length for each of these operations.

Chinese Remainder Theorem. Let $N = \prod_{i=1}^{\ell} P_i^{\alpha_i}$. The Chinese remainder theorem states that the following group isomorphism is true:

$$\mathbb{Z}_N^* \equiv \mathbb{Z}_{P_1^{\alpha_1}}^* \times \cdots \times \mathbb{Z}_{P_\ell^{\alpha_\ell}}^* \ .$$

That is, there is an isomorphism ϕ that maps $\mathbb{Z}_N^*$ to a direct product of the groups $\mathbb{Z}_{P_i^{\alpha_i}}$. This isomorphism is efficiently computable in both directions.

Slightly less abstractly, ϕ maps every element $x \in \mathbb{Z}_N^*$ as into a tuple of elements

$$\vec{x} = (x \pmod{P_1^{\alpha_1}}, x \pmod{P_2^{\alpha_2}}, \ldots, x \pmod{P_\ell^{\alpha_\ell}}) \ .$$

Multiplying x and y mod N is equivalent to multiplying $\vec{x}$ and $\vec{y}$ componentwise. ϕ is a one-to-one onto mapping. ϕ^{-1} can be computed as a linear function. Letting $\vec{x} = (x_1, \ldots, x_\ell)$, it turns out that

$$\phi^{-1}(\vec{x}) = \sum_{i=1}^{\ell} c_i x_i \pmod N$$

where, letting Q_i denote $P_i^{\alpha_i}$ and letting Q_{-i} denote $\prod_{j \neq i} P_j^{\alpha_j}$, the chinese remainder coefficients c_i are as follows:

$$c_i = Q_{-i} \cdot (Q_{-i}^{-1} \pmod{Q_i})$$

I will leave it as an exercise to check that this is indeed the inverse of ϕ.

3 The Multiplicative Group $\mathbb{Z}_N^*$

The multiplicative group of numbers mod N, denoted $\mathbb{Z}_N^*$, consists of the set

$$S = \{1 \leq a < N : \gcd(a, N) = 1\}$$

with multiplication mod N being the group operation.

Theorem 3.1. *$\mathbb{Z}_N^*$ is a group. Operations in this group including multiplication and computing inverses can be done in time polynomial in the bit length of the numbers, namely* $\mathsf{poly}(\log N)$.

Proof. Exercise. □

Examples.

$$\mathbb{Z}_2^* = \{1\}$$
$$\mathbb{Z}_3^* = \{1, 2\}$$
$$\mathbb{Z}_4^* = \{1, 3\}$$
$$\mathbb{Z}_5^* = \{1, 2, 3, 4\}$$
$$\mathbb{Z}_6^* = \{1, 5\}$$
$$\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$$

Some further facts about $\mathbb{Z}_N^*$:

- The order of $\mathbb{Z}_N^*$, the number of positive integers smaller than N that are relatively prime to it, is called the Euler totient function of N denoted $\varphi(N)$.
- If $N = \prod_i p_i^{\alpha_i}$ is the prime factorization of N, then

  $$\varphi(N) = \prod_i p_i^{\alpha_i - 1}(p_i - 1)$$

  Prove this! For example, if N is prime, then $\varphi(N) = N - 1$ and if $N = PQ$ is a product of two primes, then $\varphi(N) = (P - 1)(Q - 1)$.
- For every $a \in \mathbb{Z}_N^*$,

  $$a^{\varphi(N)} = 1 \pmod{N}$$

  This is called Euler's theorem, a direct consequence of Lagrange's theorem and the fact that the order of $\mathbb{Z}_N^*$ is $\varphi(N)$. In the special case where the modulus P is prime, for every $a \in \mathbb{Z}_P^*$,

  $$a^{P-1} = 1 \pmod{P}$$

  a fact that is referred to as Fermat's little theorem.

4 The Multiplicative Group $\mathbb{Z}_P^*$ for a Prime P

Let's first focus on the case of $\mathbb{Z}_P^*$ when P is prime. Then, $\mathbb{Z}_P^* = \{1, 2, 3, \ldots, P - 1\}$ and its order is $\varphi(P) = P - 1$.

$\mathbb{Z}_P^*$ is Cyclic. The following is a very important property of $\mathbb{Z}_P^*$ when P is prime.

Theorem 4.1. *If P is prime, then $\mathbb{Z}_P^*$ is a cyclic group.*

For example, $\mathbb{Z}_7^* = \{1,2,3,4,5,6\} = \{5^6, 5^4, 5^5, 5^2, 5^1, 5^3\} = \{5^i \pmod 7 : i > 0\}$. So, 5 is a generator of $\mathbb{Z}_p^*$. We refer the reader to Angluin's notes for a proof.

It is very tempting to try to prove this theorem by appealing to Theorem 1.2 which says that every group with prime order is cyclic. Many before you have succumbed to this mistake. Be careful, and note that the order of $\mathbb{Z}_P^*$ is $P-1$, which is *decidedly not prime*.

Even if N is not prime, $\mathbb{Z}_N^*$ may end up being cyclic: in particular, it is known that this happens exactly when $N = 1, 2, 4, p^k$ or $2p^k$ where p is an odd prime number. However, we will never encounter these beasts in our course.

One could ask several followup questions such as:

- how many generators are there for $\mathbb{Z}_p^*$?
- how to tell (efficiently) if a given element g is a generator?
- how to sample a random generator for $\mathbb{Z}_p^*$?

We will answer them in the sequel, starting with the first question.

$\mathbb{Z}_P^*$ has lots of generators. The proof of the following theorem is contained in the proof of theorem 4.1 and can be found in Angluin's notes.

Theorem 4.2. *The number of generators in $\mathbb{Z}_P^*$ is $\varphi(P-1)$.*

Now, how large is $\phi(P-1)$ asymptotically? This is answered by the following classical theorem.

Theorem 4.3. *For every integer N, $\phi(N) = \Omega(N/\log\log N)$.*

In other words, if you pick a random element of $\mathbb{Z}_P^*$, you will see a generator with probability

$$\varphi(P-1)/(P-1) = \Omega(1/\log\log P)$$

which is polynomial in $1/\log P$. So, reasonably often!

The multiplicative group $\mathbb{Z}_P^*$ and the additive group $\mathbb{Z}_{P-1}$. Let us note the following structural fact about $\mathbb{Z}_P^*$ before proceeding further. These two groups are isomorphic with an isomorphism ϕ that maps $x \in \mathbb{Z}_{P-1}$ to $g^x \in \mathbb{Z}_P^*$. In particular, consider

$$\phi(x) = g^x \pmod P$$

we have $\phi(x+y) = \phi(x) \cdot \phi(y)$.

The isomorphism is efficently computable in the forward direction (exponentiation, using the repeated squaring algorithm) but not known to be efficiently computable in the reverse direction. The latter is the **discrete logarithm problem**.

Here is another quick application of this isomorphism:

Lemma 4.4. *Let P be an odd prime. If g is a generator of $\mathbb{Z}_P^*$, then so is g^x as long as x and $P-1$ are relatively prime.*

Proof. Exercise. □

As a corollary, we immediately derive the fact that $\varphi(P-1)$ elements of $\mathbb{Z}_P^*$ are generators.

Primes, Primality Testing and Generating Random Primes. The prime number theorem tells us that there are sufficiently many prime numbers. In particular, letting $\pi(N)$ denote the number of prime numbers less than N, we know that

$$\pi(N) = \Omega(N/\log N)$$

Thus, if you pick a random number smaller than N, with probability $1/\log N$ (which is 1/polynomial in the bit-length of the numbers in question) you have a prime number at hand.

The next question is how to recognize that a given number is prime. This has been the subject of extensive research in computational number theory with many polynomial-time algorithms, culminating with the deterministic polynomial-time primality testing algorithm of Agrawal, Kayal and Saxena (AKS) in 2002.

These two facts put together tell us how to generate a random n-bit prime number — just pick a random number less than 2^n and test if it is prime. In expected n iterations of this procedure, you will find a n-bit prime number, even a random one at that.

How to tell if a given g is a generator of $\mathbb{Z}_P^*$? We know that $g^{P-1} = 1 \pmod P$ and we want to check if there is some smaller power of g that equals 1. We also know (by Lagrange) that any such power has to be a divisor of $P-1$. However, there are a large number of divisors of $P-1$, roughly $P^{1/\log\log P}$ which is not polynomial (in $\log P$.) It turns out, however, that you do not need to check all divisors, but rather only the *terminal* divisors.

More precisely, let $P-1 = \prod_i q_i^{\alpha_i}$ be the prime factorization of $P-1$. Then, the following algorithm works, on input g and the prime factorization of $P-1$:

1. For each i, check if $g^{(P-1)/q_i} = 1 \pmod P$. If yes, say "not a generator" and otherwise say "generator".

That's nice. But can one tell if g is a generator given only g and P (as opposed to the prime factorization of $P-1$ which is in general hard to compute)? We don't know, so we have to find a way around it. There are two solutions:

Solution 1. Pick $P = 2Q+1$ where Q is prime. Such primes are called safe primes, and Q is called a Sophie-Germain prime after the famous mathematician. While there are infinitely many primes, it has only been conjectured that there are infinitely many Sophie-Germain primes. This remains unproven.

Solution 2. Pick a random P *together with its prime factorization.* This, it turns out, can be done due to a clever algorithm of Kalai. (reference on the webpage)

We will more or less always stick with Solution 1 in this course.

5 One-way Functions, PRGs and PRFs from Discrete Logarithms

A one-way function is a function f that is easy to compute but hard to invert *on average*. A formal definition will come in later lectures, but for now, let us present an informal candidate.

$$f(P, g, x) = (P, g, g^x \pmod P)$$

Computing this function can be done in time polynomial in the input length. However, inverting is the discrete logarithm problem (defined formally below) which is conjectured to be hard.

> **Discrete Log Assumption (DLOG): For a random n-bit prime P and random generator g of $\mathbb{Z}_P^*$, and a random $x \in \mathbb{Z}_{P-1}$, there is no polynomial (in n) time algorithm that computes x given $P, g, g^x \pmod P$.**

In fact, this is not just a one-way function, but can also be made into a family of one-way *permutations*. More on that later. As we will see in a couple of lectures, one-way permutations can be used to build pseudo-random generators; and as we saw already, pseudorandom generators can be used to build pseudorandom functions and stateless secret-key encryption and authentication. So, we can do all the crypto we saw so far based on the hardness of the discrete logarithm problem.

However, going via this route may not be the most efficient. So, we will look at related problems and try to build more efficient PRGs and PRFs.

5.1 Algorithms for the Discrete Log Problem

We will present an algorithm for discrete logarithms that works over any group. This is the so-called *baby-step giant-step algorithm* that runs in time $O(\sqrt{|G|})$. It is still exponential but a square-root factor faster than naïve exhaustive search.

The Idea. Given elements $g, h \in G$, the problem is to find an x such that $h = g^x$ in G. Since we know that $0 \leq x < |G|$, let us write x as

$$x = x_1 P + x_0$$

where P is an integer that is close to $\sqrt{|G|}$, and $0 \leq x_0, x_1 < P$. In other words, write x in base-P. Now, we know that

$$h = g^{x_1 P + x_0} = (g^P)^{x_1} \cdot g^{x_0}$$

or

$$h \cdot g^{-x_0} = (g^P)^{x_1}$$

Conversely, any such pair of integers (x_0, x_1) will solve the discrete log problem for us. So, let's create two tables, that enumarerate the values of hg^{-x_0} and $(g^P)^{x_1}$ respectively, as follows.

$$\text{Table } 0: \begin{bmatrix} (0, hg^{-0}) \\ (1, hg^{-1}) \\ (2, hg^{-2}) \\ \cdots \\ (P-1, hg^{-(P-1)}) \end{bmatrix} \quad \text{and} \quad \text{Table } 1: \begin{bmatrix} (0, (g^P)^0) \\ (1, (g^P)^1) \\ (2, (g^P)^2) \\ \cdots \\ (P-1, (g^P)^{(P-1)}) \end{bmatrix}$$

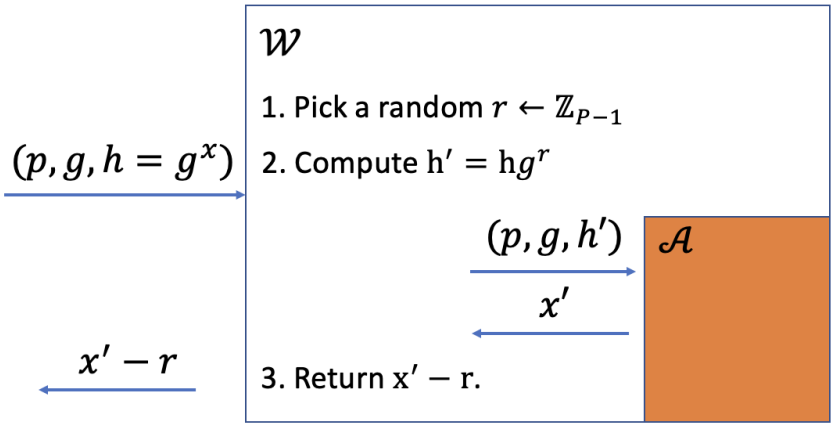

Figure 2: The toy version of random self-reduction for discrete log that randomizes only h.

Let us now restate the job of the algorithm: find two entries, one in each table, with a common second component. This can be done in time $O(P \log P) = O(\sqrt{|G|} \log |G|)$ by sorting table 0 and table 1 by their second components (in time $O(P \log P)$) and then running through them to find a common element using the two-finger algorithm (remember Merge-Sort?) in $O(P)$ time.

This is still the best known algorithm for computing discrete logarithms in arbitrary groups, and even in some practically important ones such as elliptic curve groups. However, for $\mathbb{Z}_P^*$, better algorithms are known: the index-calculus algorithm runs in $2^{O(\sqrt{\log P \log\log P})}$ time, and the number field sieve algorithm that runs in $2^{O((\log P)^{1/3}(\log\log P)^{2/3})}$ time. Both are *sub*-exponential, but far from a polynomial-time algorithm.

5.2 Random Self-Reducibility

How hard is the discrete logarithm problem for different choices of P, g and x? Could it be that the problem is hard for a worst-case choice of these values, yet easy for many values, or random values? What we would ideally like is a *worst-case to average-case reduction* that tells us that if the discrete logarithm problem can be solved for (appropriately defined) random p, g and x, then it can be solved for *every* p, g and x. We unfortunately do not know such a statement, but we can prove *something*.

As a warmup, fix P and a generator g of $\mathbb{Z}_P^*$, and assume that you have an algorithm $\mathcal{A}$ that solves discrete log for $(P, g, g^x \pmod P)$ for a random P. How do you construct out of $\mathcal{A}$ an algorithm $\mathcal{W}$ that solves discrete log for *every* $(P, g, g^x \pmod P)$? The idea is to turn this instance of the discrete log problem into a random instances in such a way that a solution to the random instance can be mapped back into a solution for the given instance. The reduction is shown in Figure 2. The more general theorem is given below.

Theorem 5.1 (Random Self-Reducibility). *Fix a prime P. If discrete log over $\mathbb{Z}_P^*$ can be solved in polynomial-time for a uniformly random generator g and $x \in \mathbb{Z}_{P-1}$, then it can be solved*

for every generator g and $x \in \mathbb{Z}_{P-1}$.

Proof. Establishing the theorem requires showing a reduction. Assume $\mathcal{A}$ (for average-case) is a $\mathsf{poly}(n)$-time algorithm that solves discrete log over $\mathbb{Z}_P^*$ for a $1/\mathsf{poly}(n)$ fraction of (g, x) where $n = \log P$. We wish to demonstrate a $\mathsf{poly}(n)$-time algorithm $\mathcal{W}$ (for worst-case) that solves discrete log over $\mathbb{Z}_P^*$ for every (g, x).

The algorithm $\mathcal{W}$, on input $(P, g, h = g^x \pmod P)$, works as follows:

- Pick a random s s.t. $\gcd(s, P-1) = 1$. Let $g' = g^s \pmod P$.
- Pick a random $r \leftarrow \mathbb{Z}_{P-1}$ and compute $h' = h^s \cdot g^r \pmod P$.
- Return $x = s^{-1}(x' - r) \pmod{P-1}$ as the discrete log solution to (P, g, h).

First, note that by our choice of s, g' is a generator as well. Indeed, it is a uniformly random generator of $\mathbb{Z}_P^*$. Now

$$h' = h^s \cdot g^r = g^{sx+r} \pmod P$$

which, by the random choice of r, is a uniformly random element of $\mathbb{Z}_P^*$ (even conditioned on g').

Thus, $\mathcal{A}$ will produce the discrete log x' of h' w.r.t. g' with probability $1/\mathsf{poly}(n)$. In this event, we can compute x as

$$x = s^{-1}(x' - r) \pmod{P-1}$$

If $\mathcal{A}$ refuses to solve the discrete log problem or it outputs an incorrect answer (either of which can happen with probability at most $1 - 1/\mathsf{poly}(n)$), then $\mathcal{W}$ repeats this process again with fresh choice of s and r. Thus the reduction succeeds with probability $1 - \mathsf{negl}(n)$ when run on $n \cdot \mathsf{poly}(n)$ trials.

□

5.3 Quadratic Residues

Let us take a detour and look at a different facet of the equation

$$h = g^x \pmod P$$

namely the problem of computing g given h and x. In fact, we will look at the special case of $x = 2$ for now. So, can you compute square roots mod P?

Before getting there, one could ask which numbers mod P are squares, namely which $h \in \mathbb{Z}_P^*$ can be written as $h = g^2 \pmod P$ for some $g \in \mathbb{Z}_P^*$? Such numbers are also called *quadratic residues*. Can you efficiently determine, given h, which it is a quadratic residue?

Theorem 5.2. *Let P be an odd prime number and g be some generator of $\mathbb{Z}_P^*$. The following conditions are equivalent.*

1. $\mathsf{dlog}_g(h)$ *is an even number;*
2. *h is a quadratic residue mod P; and*

3. $h^{(P-1)/2} = 1 \pmod{P}$.

Proof. We will show the following implications.
(1)⇒(2). Assume $h = g^x \pmod{P}$ where x is an even number. Then, $h = (g^{x/2})^2$, so it is a quadratic residue.
(2)⇒(3). Assume $h = t^2 \pmod{P}$ is a quadratic residue. Then,

$$h^{(P-1)/2} = t^{P-1} = 1 \pmod{P}$$

where the second equation is by Fermat's little theorem (or Lagrange's theorem).
(3)⇒(1). Assume $h^{(P-1)/2} = 1 \pmod{P}$. Letting $h = g^x$, we have

$$g^{x(P-1)/2} = 1 \pmod{P}$$

This means that $P - 1$ divides $x(P-1)/2$ which can only happen when x is even. □

In other words, exactly half the elements of $\mathbb{Z}_P^*$ are quadratic residues. If t is a square root of h, so is $-t$. Thus, every quadratic residue has two distinct square roots. One can compute the two square roots of any given number in poly(n) = poly$(\log P)$ time. We will prove this below, for a special class of P's.

Theorem 5.3. *Assume $P = 3 \pmod{4}$. The two square roots of h mod P are $h^{(P+1)/4}$ and $-h^{(P+1)/4}$.*

Proof. First of all, $(P+1)/4$ is an integer precisely because $P = 3 \pmod{4}$. Now, assume that $h = t^2 \pmod{P}$. Then,

$$h^{(P+1)/4} = t^{(P+1)/2} = t \cdot t^{(P-1)/2} = \pm t \pmod{P}$$

The last equality is because $t^{P-1} = 1 \pmod{P}$ and $t^{(P-1)/2}$, being its square root, is either 1 or -1 mod P. □

For the case of general P, a beautiful algorithm due to Berlekamp computes square roots in probabilistic polynomial time.

5.4 Other Roots

If e is relatively prime to $P - 1$, it is much easier to solve the equation $h = g^e \pmod{P}$ given h and e. It is not hard to verify that if $d = e^{-1} \pmod{P-1}$ (this exists because e is relatively prime to $P - 1$), then

$$g = h^d \pmod{P}$$

is the unique solution. This will come in handy when we describe the RSA cryptosystem later in the course.

5.5 The Diffie-Hellman Assumptions

Given g^x and g^y mod P, you can compute $g^{x+y} = g^x \cdot g^y \pmod P$, but can you compute $g^{xy} \pmod P$? If you can compute discrete logarithms, then you can compute x from g^x, and raise g^y to x to get $(g^y)^x = g^{xy} \pmod P$. But discrete log is hard, so this isn't an efficient way to solve the problem.

Indeed, this problem, called the computational Diffie-Hellman (CDH) problem, appears to be computationally hard, in fact as hard as computing discrete logarithms!

> **Computational Diffie-Hellman Assumption: For a random n-bit prime P and random generator g of $\mathbb{Z}_P^*$, and random $x, y \in \mathbb{Z}_{P-1}$, there is no polynomial (in n) time algorithm that computes $g^{xy} \pmod P$ given $P, g, g^x \pmod P, g^y \pmod P$.**

Moreover, it appears hard to even tell if you are given the right answer or not! But this requires some care to formalize. At first, one may think that given P, g, g^x, g^y, it is hard to distinguish between the right answer $g^{xy} \pmod P$ versus a random number u mod P. Let us call the assumption that this decisional problem is hard the *decisional Diffie-Hellman* (DDH) assumption.

> **Decisional Diffie-Hellman Assumption (first take): For a random n-bit prime P and random generator g of $\mathbb{Z}_P^*$, and random $x, y \in \mathbb{Z}_{P-1}$ and a random number $u \in \mathbb{Z}_P^*$, there is no polynomial (in n) time algorithm that distinguishes between $(P, g, g^x \pmod P, g^y \pmod P, g^{xy} \pmod P)$ and $(P, g, g^x \pmod P, g^y \pmod P, u \pmod P)$.**

However, this assumption turns out to be false as we will now show.

5.5.1 DDH is False in $\mathbb{Z}_P^*$

Let's step back and ask if the DDH assumption is actually true. It seems awfully strong on first look. It says not only that it is hard to compute g^{xy} from g^x and g^y, but also that not even a single bit of g^{xy} can be computed (with any polynomial advantage beyond trivial guessing).

We will now show that some information about g^{xy} indeed does leak from g^x and g^y. For this end, we will use the notion of quadratic residues in $\mathbb{Z}_P^*$ (or, perfect squares mod P). Recall by the discussion in the previous section that

- exactly half the elements in $\mathbb{Z}_P^*$ are quadratic residues,
- these are precisely the elements h whose discrete logarithm w.r.t. (some generator) g is *even*, and
- we can decide in polynomial-time whether h is a quadratic residue or not.

Notice what this says: even though the discrete log x itself is hard to compute in its entirety, its parity can in fact be efficiently computed!

Armed with this observation, we prove the following theorem.

Problem	$\mathbb{Z}_N^*$, prime N	$\mathbb{Z}_N^*$, composite N
Powering: Given $N, g \in \mathbb{Z}_N^*, x \in Z_{\varphi(N)}$, compute $h = g^x \pmod N$	poly-time	poly-time
Discrete Logarithm: Given N and $g, h \in \mathbb{Z}_N^*$, compute x s.t. $h = g^x \pmod N$	hard	hard
Root-Finding: Given N and $g \in \mathbb{Z}_N^*$, $x \in \mathbb{Z}_{\varphi(N)}$, compute g s.t. $h = g^x \pmod N$	poly-time (randomized)	hard

Figure 3: The complexity of three related problems.

Theorem 5.4. *g^{xy} is a quadratic residue if and only if either g^x or g^y (or both) is a quadratic residue.*

Proof. g^x (resp. g^y) is a quadratic residue if and only if x (resp. y) is even (and therefore even mod $P-1$ since $P-1$ is itself necessarily even). Thus, if g^x or g^y is a quadratic residue, then $xy \pmod{P-1}$ is even and therefore g^{xy} is a quadratic residue as well. Conversely, if g^{xy} is a quadratic residue, then xy is even, so either x or y is even. In other words, either g^x or g^y is a quadratic residue. This finishes the proof. □

Thus, given g^x and g^y, one can tell whether either of them is a quadratic residue and therefore whether g^{xy} *should be* a quadratic residue. This immediately translates to an algorithm that distinguishes between g^{xy} and a uniformly random element mod P.

Thus, we need to refine our assumption. Looking at the core reason behind the above attack, we see that there is a $1/2$ chance that g^x falls into a subgroup (the subgroup of quadratic residues, to be precise) and once that happens, g^{xy} is also in the subgroup no matter what y is. These properties are furthermore detectable in polynomial-time which led us to the attack.

A solution, then, is to work with **subgroups of $\mathbb{Z}_P^*$ of prime order**. In particular, we will take $P = 2Q + 1$ to be a safe prime and work with $\mathcal{QR}_P$, the subgroup of quadratic residues in $\mathbb{Z}_P^*$. Recall that the subgroup has order $(P-1)/2 = Q$ which is indeed prime! By virtue of this, every non-identity element of $\mathcal{QR}_P$ is its generator. With this change, we can state the following DDH assumption which is widely believed to be true.

> **Decisional Diffie-Hellman Assumption (final): Let $P = 2Q + 1$ be a random n-bit safe prime and let $\mathcal{QR}_P$ denote the subgroup of quadratic residues in $\mathbb{Z}_P^*$. For a random generator g of $\mathcal{QR}_P$, and random $x, y \in \mathbb{Z}_Q$ and a random number $u \in \mathcal{QR}_P$, there is no polynomial (in n) time algorithm that distinguishes between $(P, g, g^x \pmod P, g^y \pmod P, g^{xy} \pmod P)$ and $(P, g, g^x \pmod P, g^y \pmod P, u \pmod P)$.**

We know that **DLOG** $\rightarrow$ **CDH** $\rightarrow$ **DDH** but no implications are known in the reverse directions.

6 PRGs and PRFs from the Diffie-Hellman Assumption

PRG from DDH. Here is a candidate PRG whose pseudorandomness follows from the DDH assumption:

$$G(P, g, x, y) = (P, g, g^x, g^y, g^{xy})$$

where $P = 2Q + 1$ is a safe prime, g is a generator of the prime-order group $\mathcal{QR}_P$, and $x, y \leftarrow \mathbb{Z}_Q$.

Indeed, this expands two group elements into three. We can also speak about a family of pseudorandom generators which turns out to be more convenient. The family is indexed by P and g and works as follows:

$$G_{P,g}(x, y) = (g^x, g^y, g^{xy})$$

The security definition now says that given a random function chosen from the family $\{G_{P,g} : P = 2Q + 1, P \text{ and } Q \text{ prime and } g \text{ a generator of } \mathcal{QR}_P\}$, it is hard to distinguish between g^x, g^y, g^{xy} and a sequence of three random group elements in $\mathcal{QR}_P$.

PRF from DDH. One could start from the PRG above and construct a PRF using the GGM construction. But there are more direct ways to do this by exploiting the number-theoretic structure in the function.

Here is a candidate PRF due to Naor and Reingold whose pseudorandomness follows from the DDH assumption. The key is a generator g for $\mathcal{QR}_P$ and a vector $\vec{x} = (x_1, \ldots, x_\ell) \in \mathbb{Z}_Q^\ell$ and the public description of the family is P. The PRF is defined as:

$$F_{P,g,\vec{x}}(a) = g^{\prod_{j=1}^{\ell} x_j^{a_j}}$$

In other words, it computes a subset product in the exponent.

Theorem 6.1 (Naor and Reingold, FOCS 1997)**.** *Let $P = 2Q + 1$ be a safe prime and let g be a generator of $\mathcal{QR}_P$. The family of functions $\{F_{P,g,\vec{x}} : \vec{x} \in \mathbb{Z}_Q^\ell\}$ is a pseudorandom function family.*

Proof. The proof will go very much in the same vein as that of the GGM tree-based PRF. Assume that there is a ppt adversary $\mathcal{A}$ that has oracle access to either $F_{P,g,\vec{x}}$ for a random $\vec{x}$ or a uniformly random function from $\{0,1\}^\ell$ to $\mathcal{QR}_P$. Let us define hybrid experiments $H_0, H_1, \ldots, H_\ell$ as follows:

H_0**:** In this hybrid, $\mathcal{A}$ gets oracle access to $F_{P,g,\vec{x}}$.

H_1**:** In this hybrid, we pick two elements $z_0 = 1$ and z_1 uniformly at random and answer each query $\vec{a}$ with

$$g^{z_{a_1} \cdot \prod_{j>1} x_j^{a_j}}$$

H_2**:** In this hybrid, we pick four elements $z_{00} = 1, z_{01}, z_{10}, z_{11}$ uniformly at random and answer each query $\vec{a}$ with

$$g^{z_{\vec{a}[2]} \cdot \prod_{j>2} x_j^{a_j}}$$

where $\mathbf{a}[i]$ denotes the first i bits of $\vec{a}$.

...

H_i: In this hybrid, we pick 2^i elements z_α for every $\alpha \in \{0,1\}^i$ at random except with $z_{0^i} = 1$ (later, we will see how to implicitly pick them so as to not run in time 2^i) and answer each query $\vec{a}$ with

$$g^{z_{\vec{a}[i]} \cdot \prod_{j>i} x_j^{a_j}}$$

H_{i+1}: In this hybrid, we pick 2^{i+1} elements z_α for every $\alpha \in \{0,1\}^{i+1}$ at random except with $z_{0^{i+1}} = 1$ and answer each query $\vec{a}$ with

$$g^{z_{\vec{a}[i+1]} \cdot \prod_{j>i+1} x_j^{a_j}}$$

...

H_ℓ: In this hybrid, we pick 2^ℓ elements z_α for every $\alpha \in \{0,1\}^\ell$ at random except with $z_{0^\ell} = 1$ and answer each query $\vec{a}$ with

$$g^{z_{\vec{a}}}$$

The adversary has access to the PRF in H_0 and she has access to a random function in H_ℓ. By the hybrid argument, if she can distinguish between them, she can also distinguish between H_i and H_{i+1} for some $i \in \{0, \ldots, \ell - 1\}$. We will show how to turn such a distinguisher into a breaker for the DDH assumption.

To get a bit of intuition, imagine for a moment that $\ell = 2$. Let us look at the PRF evaluated on all inputs $00, 01, 10$ and 11. This gives us four answers

$$g, g^{x_1}, g^{x_2}, g^{x_1 x_2}$$

These four elements are indistinguishable from random by the DDH assumption.

This intuition can be carried out to take advantage of a distinguisher between H_i and H_{i+1} in pretty much exactly the same way. One should be careful to generate the z_α only when warranted, using lazy evaluation, exactly as in the GGM proof. □