3 The Discrete Logarithm Problem and Public Key Encryption

Now that we've laid a theoretical groundwork, we can look at the types of encryption algorithms that are in widespread use today. This section will combine two big ideas: we will do *asymmetric encryption* by leveraging the mathematical difficulty of the *discrete logarithm problem.*

3.1 Key Exchange

One of the major limitations on the algorithms we've discussed—and many we haven't—is the problem of *key exchange.* Any symmetric encryption algorithm requires the two parties to exchange a key in secret. This can often be difficult, since the whole point of cryptography is to protect your communications from evesdroppers.

Modern algorithms allow us to exchange keys in public, in a way that does not give evesdroppers access to the key even if they can overhear and understand the message. An algorithm that enables this is called a key exchange algorithm.

We illustrate the basic idea with a physical analogue. Suppose you want to send presents to a friend, but need to send them inside a locked box—otherwise the presents will be stolen out of the box during transit. This is a problem, since you don't have any locks that your friend has a key to. So you need to find a way to send the friend a key—without the key itself being stolen out of the locked box.

One solution is to pack a key inside a box that can be locked with two separate padlocks. Lock one, and then send the box to your friend. Your friend can't open the box, but they can put their own padlock on the box. Now neither of you can open it alone.

Your friend sends the box back to you, and you remove your lock. Send the box back to your friend, and they can remove their own lock and now retrieve the key you originally packed. You have successfully exchanged a key without ever sending it unlocked, despite the fact that you and your friend originally did not have a shared key.

Our task for the day will be to find a mathematical way to implement this idea.

3.1.1 Merkle's Puzzles

The first variant of this idea was developed by Ralph Merkle in 1974. He suggested the following algorithm:

Algorithm 3.1.

1. Bob generates N different symmetric keys and attaches an identification code i to each of them.
2. For each key, Bob encrypts a message of the form "This is the ith key on my list. The key is K_i." He uses an encryption algorithm that is possible but computationally expensive to brute force.
3. Bob sends Alice all N of the messages generated this way. Alice chooses one at random and brute-force decrypts it, and sends the identifier to Bob.
4. Bob and Alice can now communicate using the symmetric key they have agreed on.

Notice that Alice only needs to brute-force decrypt *one* of the messages Bob sends. However, if Eve does not know which message Alice chose, so if she intercepts Bob's transmission, she must decrypt all of them to find the identifiers and know the¡ key Alice and Bob are using.

However, while Eve's job here is harder than Alice's, it's not enough harder—it's at most quadratically harder, which isn't generally considered a good enough speed advantage for secure cryptography.

3.2 Diffie-Hellman key exchange

Diffie-Hellman key exchange is an algorithm first published by Diffie and Hellman in 1976. (It was actually discovered in 1975 by James H. Ellis, Clifford Cocks, and Malcolm J. Williamson of British intelligence, but their discovery wasn't declassified until 1997).

Algorithm 3.2. Alice and Bob wish to exchange a key. They follow the following steps:

1. Choose a large prime p, and a non-zero integer $g \in \mathbb{Z}/p\mathbb{Z}^\times$.
2. Alice chooses a secret integer a, and Bob chooses a secret integer b. Neither party reveals this integer to anyone.
3. Alice computes $A \equiv g^a \mod p$ and Bob computes $B \equiv g^b \mod p$, and they (publicly) exchange these values with each other.
4. Now Alice computes $A' \equiv B^a \mod p$ and Bob computes $B' \equiv A^b \mod p$.
5. $A' \equiv B' \mod p$, so Alice and Bob use this shared information as their key.

Remark 3.1. Technically this doesn't exchange a key so much as create a commonly known key. But it does allow Alice and Bob to mutually share information to get a mutual key, without sharing the key with evesdroppers.

Proposition 3.2. $A' \equiv B' \mod p$.

Proof. $A' \equiv B^a \equiv (g^b)^a \equiv (g^a)^b \equiv A^b \equiv B' \mod p$. □

Example 3.3. A toy example: Suppose Alice and Bob have chosen the prime 29 and the primitive root 2. (These are terrible choices for security). Alice chooses a secret key $a = 7$ and Bob chooses a secret key $b = 17$. Then Alice computes $A = g^a = 2^7 \equiv 12 \mod 29$; and Bob computes $B = g^b = 2^{17} \equiv 21 \mod 29$.

Alice sends $A = 12$ to Bob, and Bob sends $B = 21$ to Alice. Eve can observe both these numbers, which are public knowledge.

Then Alice computes $A' = B^a = 21^7 \equiv 12 \mod 29$, and Bob computes $B' = A^b = 12^{17} \equiv 12 \mod 29$. So Alice and Bob have a shared secret of 12.

Example 3.4. Suppose Alice and Bob are working with the prime modulus $p = 941$ and primitive root $g = 627$. Alice chooses a secret key $a = 347$, computes $A = g^a = 627^{347} \equiv 390 \mod 941$; and Bob chooses $b = 781$ and compute $B = g^b = 627^{781} \equiv 691 \mod 941$.

Alice sends the number 390 to Bob, and Bob sends the number 691 to Alice. Both of these numbers are visible to Eve and are public knowledge. Then Alice computes

$$A' = B^a = 691^{347} \equiv 470 \mod 941.$$

And Bob computes

$$B' = A^b = 390^{781} \equiv 470 \mod 941.$$

Now Alice and Bob have a secret number 470 in common that Eve doesn't know.

So how secure is this, really? Remember that for any sort of encryption algorithm, we want it to be easy for Alice and Bob to compute, but expensive for Eve to compute. We first need to talk about what it means for a computation to be easy or hard.

3.2.1 Security of Diffie-Hellman

First let's look at the computation Alice and Bob need to do. Each of them needs to do exponentiations: Alice first computes g^a and then she computes B^a. (Bob computes g^b and A^b, which is exactly the same process, so we'll just look at Alice for simplicity).

So how do we do a large exponentiation mod p? The obvious and naive method is to compute $g, g^2, g^3, g^4, \ldots, g^a$. So in our toy example 3.3 Alice would compute

$$\begin{array}{llll} 2^1 = 2 & 2^2 = 4 & 2^3 = 8 & 2^4 = 16 \\ 2^5 = 32 \equiv 3 & 2^6 \equiv 6 & 2^7 \equiv 12. & \end{array}$$

This is totally manageable when a and p are this small, but when they get bigger, this leads to a large number of multiplications. Alice will have to conduct a multiplications to compute g^a, and another a multiplications to compute B^a, so in total she will need to conduct $2a$ multiplications. This algorithm is thus $O(a)$.

But this is actually an exponential algorithm! Remember we don't care about the size of the number that is the input; we care about the number of bits. And the number of bits involved is $k \approx \log_2(a)$. Since $a = 2^k$ by definition, we see that this algorithm is $O(2^k)$, and thus exponential time.

Fortunately, Alice has an easier algorithm.

Algorithm 3.3.

1. Compute g^{2^k} for $2^k \leq a$. That is, compute $g, g^2, g^4, g^8, \ldots, g^{2^k}$. We can do this by repeated squaring, without computing intermediate powers.
2. Now express the exponent a in binary. That is, write $a = c_0 + c_1 \cdot 2 + c_2 \cdot 2^2 + \cdots + c_k 2^k$, where $c_i \in \{0, 1\}$.
3. Now we can compute

$$\begin{aligned} g^a &= g^{c_0 + c_1 \cdot 2 + c_2 \cdot 2^2 + \cdots + c_k 2^k} = g^{c_0} g^{c_1 \cdot 2} g^{c_2 \cdot 2^2} \cdots g^{c_k 2^k} \\ &= g^{c_0} (g^2)^{c_1} (g^{2^2})^{c_2} \cdots (g^{2^k})^{c_k}. \end{aligned}$$

   But we already know g^{2^i} for each i, and the c_i are all either 0 or 1 so don't involve any computation. So we only have to multiply up to k things together here.

With this algorithm, we do k squarings to compute the g^{2^i}, and we do k multiplications to finally compute g^a, so we need to do $2k$ total multiplications and this algorithm is $O(k)$. Since k is the number of bits of input, this algorithm is a linear-time algorithm.

Example 3.5. Let's see how this applies to our toy example 3.3. Alice wanted to compute $2^7 \mod 29$.

We first compute every 2^kth power of g, so we compute

$$2^1 = 2 \qquad 2^2 = 4 \qquad 2^4 = 16 \qquad 2^8 = 256 \equiv 24.$$

Then we write a in binary; that is, we write $7 = 1 + 2 + 2^2$. So we have

$$2^7 \equiv 2^{2^0} \cdot 2^{2^1} \cdot 2^{2^2} \equiv 2 \cdot 4 \cdot 16 \equiv 64 \equiv 12.$$

So Alice and Bob can do their work naively in about $O(2^k)$ time, or cleverly in $O(k)$. What about Eve?

Remember that Eve sees the numbers $A \equiv g^a \mod p$ and $B \equiv g^b \mod p$. She needs a way to find $A' = B' \equiv g^{ab} \mod p$. How can she do this?

The only *known* way to solve this problem is to compute at least one of a or b. We haven't proven that there's not a more efficient algorithm, but we don't know of one. So in practice, Eve wants to solve the following problem:

Discrete Logarithm Problem: Given a modulus p and integers g and A, find an integer x such that $g^x \equiv A \mod p$.

Once Eve has solved this problem, she has all the information that Alice does.

There is again a naive algorithm for this. Compute $g, g^2, g^3, \ldots \mod p$, and stop when you get A as an output. This algorithm will take about a multiplications. Thus the algorithm is $O(a) = O(2^k)$, and exponential.

We do in fact know a better algorithm for solving the Discrete Logarithm problem. Fortunately for cryptography purposes, it's not *much* better.

Algorithm 3.4 (Shanks's Babystep-Gianstep Algorithm)**.** Suppose we have a prime number p and a primitive root g, and an integer A, and we want to find an integer x such that $g^x \equiv A \mod p$. Then

1. let $n = 1 + \lfloor \sqrt{p} \rfloor$. Thus $n > \sqrt{p}$.
2. (Baby steps) Calculate $g^0, g^1, g^2, \ldots, g^n \mod p$. Find an inverse for $g^n \mod p$.
3. (Giant steps) Calculate $A, A \cdot g^{-n}, A \cdot g^{-2n}, \ldots, A \cdot g^{-n^2} \mod p$.
4. Find a match between these two lists, so that we have $g^i \equiv hg^{-jn}$.
5. Then $x = i + jn$ is a solution to $g^x \equiv h \mod p$.

Remark 3.6. This algorithm is named after Daniel Shanks, and was first discovered by Alexander Gelfond in 1962.

Proposition 3.7. *Shanks's algorithm solves the discrete logarithm problem in* $O(\sqrt{p} \cdot \log_2 p)$ *time.*

Proof. First we show that the algorithm is $O(\sqrt{p} \cdot \log_2 p)$. The lists in steps 2 and 3 each involve about n computations, so we have $2n \approx 2\sqrt{p}$ computations there. Each computation will less than $\log_2 p$ steps.

Step 4 requires finding matches between two lists; this is a standard computer science task, and takes about $\log_2$ the length of the lists, and thus is $O(\log_2 p)$. This winds up trivial when added to the previous steps. So the whole algorithm is $O(\sqrt{p} \cdot \log_2 p)$.

Now let's prove the algorithm works. If x is the solution to $g^x \equiv A \mod p$, we can write $x = nq + r$ for $0 \leq r < n$ by the division algorithm. Since $a \leq x < p$, we know that $q = \frac{x-r}{n} < \frac{N}{n} < n$ since $n > \sqrt{N}$.

So $g^x \equiv A \mod p$ is the same as $g^r \equiv A \cdot g^{-qn} \mod p$, restricted to $0 \leq r < n$ and $0 \leq q < n$. Our list from step 2 is a list of g^r, and our list from step 3 is a list of $A \cdot g^{-qn}$. So these lists will have a common element, and finding it will give us $x = r + qn$. $\square$

Example 3.8. 23 is prime, and 10 is a primitive root modulo 13. Let's solve $10^x \equiv 7 \mod 23$.

Following Shanks's algorithm, we see that $4 < \sqrt{23} < 5$ so we set $n = 5$.

We compute $1, 10, 10^2, 10^3, 10^4, 10^5$:

10^0	10^1	10^2	10^3	10^4	10^5
1	10	8	11	18	19

We need an inverse of $10^5 \equiv 19 \equiv -4$; we see that -6 will be an inverse modulo 23, so our inverse is -6 or 17.

Now we compute $7 \cdot (-6)^i$:

$7 \cdot (-6)^0$	$7 \cdot (-6)^1$	$7 \cdot (-6)^2$	$7 \cdot (-6)^3$	$7 \cdot (-6)^4$	$7 \cdot (-6)^5$
7	4	22	6	10	9

We find a 10 on both lists, corresponding to $r = 1$ and $q = 4$. Thus we have $x = qn + r = 4 \cdot 5 + 1 = 21$. And we check that indeed, $10^{21} \equiv 7 \mod 23$.

Example 3.9. 37 is a prime number, and 5 is a primitive root modulo 73. Let's solve $5^x \equiv 13 \mod 37$.

Following Shanks's algorithm, we see that $6 < \sqrt{37} < 7$ so we set $n = 7$.

We compute $1, 5, 5^2, \ldots, 5^7$: we get

5^0	5^1	5^2	5^3	5^4	5^5	5^6	5^7
1	5	25	14	33	17	11	18

We find an inverse of $5^7 \equiv 18$. We see that $2 \cdot 18 = 36 \equiv -1$, so the inverse of 18 is -2. Now we compute $13 \cdot 5^{-in}$ and we get

13	$13 \cdot 5^{-7}$	$13 \cdot 5^{-14}$	$13 \cdot 5^{-21}$	$13 \cdot 5^{-28}$	$13 \cdot 5^{-35}$	$13 \cdot 5^{-42}$	$13 \cdot 5^{-49}$
13	$13 \cdot -2$	$13 \cdot (-2)^2$	$13 \cdot (-2)^3$	$13 \cdot (-2)^4$	$13 \cdot (-2)^5$	$13 \cdot (-2)^6$	$13 \cdot (-2)^7$
13	11	15	7	23	28	18	1

We see that there's a matching 11 on both lists, corresponding to $r = 6$ and $q = 1$, so we have $x = 1 \cdot 7 + 6 = 13$. And we can check that $5^{13} \equiv 13 \mod 37$, as desired.

3.3 The Discrete Logarithm and the Order of an Integer

In section 3.2 we talked about the discrete logarithm problem: given a modulus p and integers g and A, find an integer x such that $g^x \equiv A \mod p$. Clearly in the specific context of breaking Diffie-Hellman, this equivalence has a solution. But when should it have a solution in general? And why do we call this a logarithm?

Recall that in the real numbers, the logarithm $\log_a(b)$ is the (unique) solution to the equation $a^x = b$. We can define an analog of this operation in modular arithmetic, after we lay some groundwork.

Definition 3.10. Let m be a positive integer. We call a number $a \in \mathbb{Z}/m\mathbb{Z}$ a *unit* modulo m if a has an inverse modulo m. We denote the set of units modulo m by $\mathbb{Z}/m\mathbb{Z}^\times$.

If p is a prime, then $\mathbb{Z}/p\mathbb{Z}^\times = \{1, 2, \ldots, p-1\}$.

Recall from section 1.4.2 that a is a unit modulo m if and only if $\gcd(a, m) = 1$. Thus if we take $m = p$ to be prime, everything becomes much nicer.

Theorem 3.11 (Fermat's Little Theorem)**.** *Let p be a prime. If $\gcd(a, p) = 1$, then $a^{p-1} \equiv 1 \mod p$.*

Sometimes, we instead say that $a^p \equiv a \mod p$. These two statements are equivalent.

Remark 3.12. This theorem was first proven by Leonhard Euler.

Every element will give us 1 if we raise it to the power of $p-1$ modulo p. Some elements don't need quite so high a power. For instance, of $p = 11$ then we have $10^2 = 100 \equiv 1 \mod 11$. But others will not reach 1 until the power gets all the way to $p-1$.

Definition 3.13. We say that a number $g \in \mathbb{Z}/p\mathbb{Z}$ is a *primitive root* modulo p if the set $\{g, g^2, g^3, \ldots, g^{p-1}\} = \mathbb{Z}/p\mathbb{Z}^\times$—that is, if raising g to successive powers gives every unit modulo p.

Necessarily, if the $p-1$ elements of $\{g, g^2, \dots, g^{p-1}\}$ hit all $p-1$ elements in $\mathbb{Z}/p\mathbb{Z}$, then there are no repetitions.

Example 3.14. 3 is a primitive root mod 7, since we have $3, 9 \equiv 2, 6, 18 \equiv 4, 12 \equiv 5, 15 \equiv 1$.

2 is not a primitive root mod 7, since we get $2, 4, 8 \equiv 1$ and we hit a repetition before we get every element of $\mathbb{Z}/7\mathbb{Z}^\times$.

But 2 is a primitive root mod 11, since we get $2, 4, 8, 16 \equiv 5, 10, 20 \equiv 9, 18 \equiv 7, 14 \equiv 3, 6, 12 \equiv 1$.

And we just saw that 10 is not a primitive root mod 11.

There's no algorithm for finding primitive roots; all you can do is keep trying until one of them works. 2 often but not always works. We can reduce the time it takes to test a number with the following fact:

Fact 3.15. *If $g \in \mathbb{Z}/p\mathbb{Z}^\times$ then $\#\{g^i \mod p : 1 \leq i \leq p-1\} | p-1$.*

Thus in particular, if we compute $g, g^2, \dots, g^{(p-1)/2}$ and we haven't found a number equivalent to 1, then we know g is a primitive root.

Definition 3.16. Let p be prime, g a primitive root mod p, and $h \in \mathbb{Z}/p\mathbb{Z}^\times$. Then if x is a natural number such that $g^x \equiv h \mod m$, we say that x is a *discrete logarithm of h to the base g modulo m.* The problem of finding a solution to this congruence is the *Discrete Logarithm Problem.*

Some authors will call this the *index* of h with respect to g, denoted $\operatorname{ind}_g(h)$.

As posed here, the Discrete Logarithm Problem always has a solution. In fact it will always have infinitely many, because if x is a solution then $x + n(p-1)$ is also a solution for any $n \in \mathbb{Z}$. We generally take "the" solution to have the property that $0 \leq x \leq p-2$. (Notice that this same problem comes up in complex analysis, where the logarithm is only defined modulo $2\pi i$).

Example 3.17. We know that 2 is a primitive root mod 11. So what is $\log_2(6)$ mod 11?

We compute $2, 4, 8, 16 \equiv 5, 10, 20 \equiv 9, 18 \equiv 7, 14 \equiv 3, 6$, so $\log_2(6) = 9$.

Example 3.18. 3 is a primitive root mod 7. What is $\log_3(4)$ mod 7?

We compute $3, 9 \equiv 2, 6, 18 \equiv 4$ so $\log_3(4) = 4$.

Example 3.19. 2 is a primitive root mod 29. What's $\log_2(7)$ mod 29?

We compute $2, 4, 8, 16, 32 \equiv 3, 6, 12, 24, 48 \equiv 19, 38 \equiv 9, 18, 36 \equiv 7$. So $\log_2(7) = 12$.

Remark 3.20. Like the problem of finding a primitive root, there isn't really a much better way to find the discrete logarithm of a number than just trying things until one works. You can get a speed up from $O(p)$ (trying all $p-2$ possibilities), to to $O(\sqrt{p})$ with clever algorithms, but that doens't help as much as you'd think against the inherent exponential growth.

The modular logarithm maintains many of the same properties that the real-number logarithm has.

Fact 3.21. *1.* $\log_g(1) = 0$

2. $\log_g(ab) \equiv \log_g(a) + \log_g(b) \mod p-1$

3. $\log_g(a^r) \equiv r\log_g(a) \mod p-1.$

Remark 3.22. Property (2) tells us that $\log_g$ is a group isomorphism from $\mathbb{Z}/p\mathbb{Z}^\times$ to $\mathbb{Z}/(p-1)\mathbb{Z}$.

3.3.1 The order of an integer

We can develop these ideas a little further to handle cases where the modulus isn't prime. But the statements are a bit more complicated.

Definition 3.23. Let m be a positive integer. We define the *Euler totient function* $\phi(m)$ to be the number of integers between 0 and m that are relatively prime to m.

There is a straightforward way to compute $\phi(m)$, but it's a bit too complicated to explain here. We will state the limited result that if p, q are primes, then $\phi(p) = p-1$ and $\phi(pq) = (p-1)(q-1)$, which we will need.

Now we can state a generalization of Fermat's little theorem 3.11:

Theorem 3.24 (Euler's Theorem). *If a, m are natural numbers and* $\gcd(a,m) = 1$, *then* $a^{\phi(m)} \equiv 1 \mod m$.

Example 3.25. Let $p = 7$ and $q = 11$. We can see that, for instance,

$$\begin{aligned}
3^6 &= 3^3 \cdot 3^3 \equiv (-1)(-1) \equiv 1 \mod 7 \\
3^{10} &= (3^2)^5 \equiv (-2)^5 \equiv -32 \equiv 1 \mod 11 \\
3^{60} &= (3^4)^{15} \equiv 4^{15} \equiv (4^3)^5 \equiv (-13)^5 \equiv -13(13^2)^2 \equiv -13(15)^2 \\
&\equiv -39 \cdot 75 \equiv -39 \cdot (-2) \equiv 1 \mod 77
\end{aligned}$$

Remark 3.26. Fermat's little theorem is a special case of Euler's theorem, and follows from the fact that if p is prime then $\phi(p) = p - 1$.

Euler's theorem implies that for any number a, there is at least one integer r such that $a^r \equiv 1 \mod m$. This allows us to give the following definition:

Definition 3.27. Let a, m be integers, and suppose $\gcd(a, m) = 1$. Then the *order* of a mod m, written $\operatorname{ord}_m(a)$, is the smallest positive integer r such that $a^r \equiv 1 \mod m$.

We observe that $1 \leq \operatorname{ord}_m(a) \leq \phi(m)$ for any a.

Example 3.28. $\operatorname{ord}_7(2) = 3$ since $2^3 = 8 \equiv 1$, and no smaller number works.

$\operatorname{ord}_7(3) = 6$ since that's the smallest power of 3 that gives us 1; we have $3, 2, 6, 4, 5, 1$.

$\operatorname{ord}_{10}(3) = 4$ since we compute $3, 9, 7, 1$.

Remark 3.29. A number g is a primitive root mod p if and only if $\operatorname{ord}_p(g) = p - 1$. More generally, g is a primitive root mod m if and only if $\operatorname{ord}_m(g) = \phi(m)$.

Fact 3.30. *$a^r \equiv a^s \mod m$ if and only if $r \equiv s \mod \operatorname{ord}_m(a)$.*

In particular, if $r \equiv s \mod \phi(m)$ then $a^r \equiv a^s \mod m$.

3.4 Public Key Cryptography

In section 3.2 we discussed a public key-exchange algorithm, in which two parties can securely exchange an encryption key over an insecure connection, so that they have access to the same key but an evesdropper does not.

This week we will study an even more significant advance: the ability to do encryption without a shared secret key at all.

3.4.1 Trapdoor functions

Recall that in section 2.3.3 we talked about one-way functions, such that f is easy to compute but f^{-1} is difficult to compute. To do asymmetric encryption well, we want something a bit more complex.

A *trapdoor function* is a function f that's easy to compute, and difficult to invert—unless you have some extra information, called *trapdoor information*, that makes inverting the function easy.

So the usual setup we'll look for will be something like this:

- A *keygen* algorithm that produces a pair of keys (k_{pub}, k_{priv}) called the *public key* and the *private key*;

- an *encryption* algorithm $e_{k_{pub}} : \mathcal{M} \to \mathcal{C}$ that uses a public key to encrypt a message; and
- a *decryption* algorithm $d_{k_{priv}} : \mathcal{C} \to \mathcal{M}$ that uses the private key to decrypt the message.

And the goal is to set this up so that knowing the public key doesn't allow you to (easily) determine the private key, and *also* that knowing the public key and the ciphertext doesn't allow you to (easily) determine the plaintext.

If Alice wants to send a message to Bob, she needs him to generate his own keypair and publish his own public key. This leads to the practice among many seriously security-minded internet users of listing a public key on their website for authentication and encrypted communications.

3.4.2 The El-Gamal Cryptosystem

The El-Gamal Cryptosystem is a public-key cryptosystem originally described by the Egyptian mathematician Taher Elgamal in 1985. Its security relies on the difficulty of computing a discrete logarithm, and acts in a sense as an extension of the Diffie-Hellman process we discussed in section 3.2.

Algorithm 3.5 (El-Gamal Cryptosystem)**.** First Alice generates a private key and a public key.

1. Choose a large prime number p, and an element $g \in \mathbb{Z}/p\mathbb{Z}$ such that $\operatorname{ord}_p(g)$ is a large prime number. This step is not at all trivial, and thus tends to be pre-standardized; everyone uses the same p and g.
2. Alice chooses a secret number a, which we call the *private key.* She does not share this number with anyone.
3. Alice computes $A \equiv g^a \mod p$, and publishes it. We call this number the *public key* because it is released to the public.

Now suppose Bob wishes to send Alice a number $2 < m < p$.

1. Bob generates a random number k, called the *ephemeral key.* This key is kept secret, and also discarded after this single message is sent.
2. Bob computes $c_1 \equiv g^k \mod p$ and $c_2 \equiv mA^k \mod p$. He sends Alice the message (c_1, c_2).

How does Alice decrypt the message? She has to use her private key a.

1. Alice computes $x \equiv c_1^a \mod p$, and then $x^{-1} \equiv c_1^{-a} \mod p$. (Alternately, she can just compute $x^{-1} \equiv c_1^{p-1-a} \mod p$ and skip computing x at all).
2. Alice then computes $c_2 x^{-1} \mod p$, which is equivalent to m.

Remark 3.31. The message is a number between 2 and m, and so takes $\log_2(m)$ bits to represent. The ciphertext consists of *two* integers between 2 and m, and thus takes $2\log_2(m)$ bits to represent. This the El-Gamal cryptosystem expands messages by a factor of two.

Remark 3.32. The requirement that $\operatorname{ord}_p(g)$ is a large prime defeats a specific attack based on something called quadratic reciprocity, which tells us whether a number is a square modulo p.

This requirement is generally met by taking the large prime p to be of the form $2q+1$ where q is also prime. We can check that any element either has order $1, 2, q$, or $2q$, and we want one with order q. Most choices will have order either q or $2q$.

If g is a primitive root mod p then $\operatorname{ord}_p(g^2) = q$. But the easiest way to find an appropriate base g for the ElGamal algorithm is to simply choose a number and raise it to the q power; if this is equivalent to 1 mod p, we have what we're looking for.

Proposition 3.33. *The decryption step of Algorithm 3.5 works. That is,* $c_2 x^{-1} \equiv m \mod p$.

Proof.

$$\begin{aligned} c_2 x^{-1} &\equiv c_2 (c_1^a)^{-1} \\ &\equiv (mA^k)(g^{ak})^{-1} \\ &\equiv (mg^{ak})g^{-ak} \\ &\equiv mg^{ak}g^{-ak} \equiv m \mod p. \end{aligned}$$

□

Example 3.34. Suppose we take $p = 467$ and $g = 4$. Alice chooses $a = 155$ as her private key, and computes $A \equiv g^a \equiv 4^{155} \equiv 43 \mod 467$. (She can do this computation using the fast exponentiation algorithm from section 3.2.1). Alice publishes the number A.

Now suppose Bob wants to send the message $m = 42$ to Alice. Bob himself chooses an ephemeral key $k = 187$. He computes:

$$\begin{aligned} c_1 &\equiv g^k \equiv 4^{187} \equiv 456 \mod 467 \\ c_2 &\equiv mA^k \equiv 42 \cdot 43^{187} \equiv 67 \mod 467. \end{aligned}$$

Bob sends Alice the message $(456, 67)$.

Alice wishes to decrypt the message. She computes:

$$\begin{aligned} x &\equiv c_1^a \equiv 456^{155} \equiv 413 \mod 467 \\ x^{-1} &\equiv c_1^{p-1-a} \equiv 147 \mod 467 \\ m &\equiv c_2 x^{-1} \equiv 67 \cdot 147 \equiv 9849 \equiv 42 \mod 467. \end{aligned}$$

Remark 3.35. Note a very important property here: Bob will send a different ciphertext depending on his random choice of k, but Alice will decrypt it to the same message regardless of Bob's choice of k. This means that there are many different ciphertexts corresponding to the same plaintext; this is why the ciphertext (which is a pair of integers) has twice as many bits as the plaintext (which is a single integer).

3.4.3 Cryptanalysis of El-Gamal

We have no ability to prove that the cryptanalysis of any reasonable algorithm is difficult, because that would effectively require proving $P \neq NP$ (and possibly more!). But we can prove that decrypting one algorithm is "at least" as hard as decrypting another. We can prove that breaking an El-Gamal cipher is at least as hard as breaking a Diffie-Hellman key exchange.

In particular, suppose Alice and Bob are doing a Diffie-Hellman key exchange, and are overheard by Eve. But Eve has an *ElGamal oracle*: a machine that will take in an ElGamal public key and ciphertext, and reveal to her the corresponding plaintext. Thus this works if Eve has any efficient way to break an ElGamal cipher—whether it involves actually finding the private key or not.

So Eve overhears Alice's transmission of $A \equiv g^a \mod p$ and $B \equiv g^b \mod p$, and she wants to compute $g^{ab} \mod p$. We saw in section 3.2.1 that there's no known efficient algorithm for doing this; the best option we have is to compute a discrete logarithm.

But with her oracle, Eve chooses a random number c_2. She tells her oracle that the public key is A and the ciphertext is (B, c_2). By definition, the oracle returns to her the "plaintext":

$$m = (c_1^a)^{-1} c_2 \equiv (B^a)^{-1} \cdot c_2 \equiv (g^{ab})^{-1} \cdot c_2.$$

Eve can then invert this number m, and compute $m^{-1} \cdot c_2 \equiv g^{ab}$ the private key that Alice and Bob have exchanged.

This doesn't tell us that breaking ElGamal is hard, because we don't know for sure that breaking Diffie-Hellman is hard. But it does prove that ElGamal is *at least* as secure as Diffie-Hellman, because if we can break ElGamal then we can also break Diffie-Hellman.

Also notice that if it's possible to break ElGamal without computing an explicit discrete logarithm, it is also possible to break Diffie-Hellman without a discrete logarithm.

3.4.4 Complexity and Implementations

Looking at the algorithm for ElGamal, we see that encrypting a message requires two exponentiations, and thus with the fast exponentiation algorithm ElGamal is $O(\log_2(p))$. Decryption requires one exponentiation, and so is also $O(\log_2(p))$.

However, while the second exponentiation in the encryption step isn't important asymptotically, it does double the amount of computation necessary to encrypt a message—and the number of bits that need to be transmitted, since a single k bit number is encrypted to be a pair of k bit numbers.

In order to save on these extra computations and bit transmissions, we often use ElGamal in a hybrid setup. The "true" message is encrypted with a *symmetric* cryptosystem, which can provide the same security for less up-front computation. Then the *key* is encrypted with the ElGamal cryptosystem.

3.5 The RSA Cryptosystem

Though ElGamal is a useful cryptosystem, it is not the first public-private key cryptosystem that was invented or published.

The RSA algorithm was published by Rivest, Shamir, and Adleman in 1978. It was first discovered by Clifford Cocks in 1973 (in conjunction with James Ellis), but this fact was not declassified by the British government until 1997.

Algorithm 3.6 (RSA Cryptosystem)**.** Suppose Alice wants to send a message to Bob.

First Bob must create and publish a public key, and compute a private key for himself.

1. Bob chooses two primes p, q and computes $N = pq$. He also computes $M = (p-1)(q-1)$.
2. Bob chooses a number e such that $\gcd(e, M) = 1$.
3. Bob publishes the pair (N, e). This is his public key. Bob does not publish p or q or M.

4. Bob computes the inverse of e modulo M, and calls it d. Thus d solves $ed \equiv 1 \mod M$. Bob's private key is the pair (M, d).

Now Alice wishes to send Bob an integer m with $1 \leq m < N$.

1. Alice computes $c \equiv m^e \mod N$. She sends c to Bob.
2. Bob computes $c^d \mod N$ and receives Alice's message m.

Proposition 3.36. *The decryption step of Algorithm 3.6 works. That is, $c^d \equiv m \mod N$.*

Proof. Recall that $ed \equiv 1 \mod (p-1)(q-1) = \phi(N)$, and thus $a^{ed} \equiv a^1 \mod N$ by 3.30.

$$\begin{aligned} c^d &\equiv (m^e)^d \\ &\equiv m^{ed} \\ &\equiv m^1 \equiv m \mod N. \end{aligned}$$

□

Example 3.37. Bob chooses the primes $p = 73$ and $q = 89$. He computes $N = pq = 6497$. He also computes $M = (p-1)(q-1) = 72 \cdot 88 = 6336$, but does not share this with anyone

Bob chooses the exponent $e = 83$ and checks that $\gcd(83, 6336) = 1$. He then computes

$$d \equiv e^{-1} \equiv 6107 \mod 6336.$$

Bob publishes the public key $(N, e) = (6497, 83)$. The pair $(M, d) = (6336, 6107)$ is his private key, which he keeps private.

Suppose Alice wishes to send Bob the message 300. Alice computes

$$c \equiv 300^{83} \equiv 4955 \mod 6497.$$

She sends Bob the message 4955.

Bob computes

$$c^d \equiv 4955^{6107} \equiv 300 \mod 6497$$

and recovers the message Alice wished to send.

Remark 3.38. These calculations are quite easy on computers with good algorithms, but are quite tedious to do by hand, especially since they tend to involve large numbers.

You can do all of this on Wolfram Alpha, including typing in "inverse of 83 mod 6336".

Example 3.39. Bob chooses the primes $p = 199$ and $q = 577$. He computes $N = pq = 114823$

3.6 Breaking RSA

In order to decrypt the ciphertext, Eve needs to solve a congruence of the form $x^e \equiv \mod N$. If Eve knows the values of p and q this is straightforward, and she can decrypt the message in exactly the same way that Bob can. As far as we know, there's no better way of breaking RSA than trying to factor N, but that doesn't mean there isn't a way we don't know about.

So how hard is factoring? It turns out to be quite difficult with current knowledge. The obvious thing to do is to just try dividing by a lot of numbers; this algorithm is about $O(N)$, which we should recall is exponential in $k = \log_2(N)$ and thus slow.

The next method is called Fermat factorization, and leverages the identity $a^2 - b^2 = (a+b)(a-b)$. Thus we can convert factoring problems into problems about writing N as a difference of squares. With some clever choices, this method works in something like $O(\sqrt[4]{N})$ time with basic optimizations—which is subexponential, but still much worse than polynomial time.

With substantial optimization, we get the Quadratic Sieve, which runs in time

$$O\left(e^{\sqrt{\log(n)\log\log(n)}}\right).$$

The best currently-known factorization algorithm for large numbers is the General Number Field Sieve, which runs in time

$$O\left(e^{\sqrt[3]{64/9}\ln(n)^{1/3}\ln(\ln(n))^{2/3}}\right).$$

This is only more efficient than the quadratic sieve for numbers larger than 10^{100}, but numbers used for cryptography are necessarily over that threshhold.

To get realistic security against modern computer-based attacks, we use moduli with about 1024 bits of entropy, which are 300-digit numbers when written in base 10. Thereare a series of standard moduli used for RSA encryption, with names like RSA-512 for the 512-bit RSA modulus.

Modern software can factor a 128-bit number on a desktop computer in less than 2 seconds, a 256-bit number in under two minutes, and a 320-bit number in under two hours.

In 2009, Benjamin Moody successfully factored RSA-512 in 73 days on a desktop computer. RSA-768 has been factored in 1500 CPU-years over two real-time years. No larger RSA number has been known to be factored. Both of these attacks used the general number field sieve.

In contrast, there's no known security advantage to using a large e instead of a small one. But it makes some people nervous, because we don't know there *isn't* an advantage.

3.6.1 Pollard's $p-1$ method

I want to explain one advanced factoring algorithm, which highlights some of the interesting features in factorization. This algorithm is only $O(\sqrt[4]{N})$, just like difference of squares and Shanks-derived approaches would be; but it points to some of the special features of factorization that make the quadratic and number-field sieves work.

We have a large number $N = pq$ and we want to find p and q. Pollard observed that if we have a large number L such that $p-1$ divides L and $q-1$ does not, then we can find p and q without too much trouble. Just as RSA encryption depends on the relationship between pq and $(p-1)(q-1)$, this algorithm will exploit that same relationship to break RSA.

In particular, we see that $L = i(p-1)$ for some i, and $L = j(q-1)+k$ for some j and some $k \neq 0$. If we assume that $p, q \nmid a$, then by Fermat's little theorem, we know that $a^L \equiv 1 \mod p$. This means that $p \mid a^L - 1$.

We also have

$$a^L = a^{j(q-1)+k} = a^k a^{j(q-1)} \equiv a^k \mod q.$$

Thus we see that it is very likely that $q \nmid a^L - 1$. And these two facts together allow us to factor N: it's easy to see that $\gcd(a^L - 1, N) = p$ in this case. But there's an efficient algorithm for computing the gcd of two numbers:

Algorithm 3.7 (Euclidean Algorithm)**.** Let a, b be integers with $a > b > 0$. We can compute $\gcd(a, b)$ by the following algorithm:

1. Set $r_0 = a$ and $r_1 = b$.
2. For each $i \geq 1$, we can divide r_{i-1} by r_i and compute the remainder term. Set r_{i+1} equal to this remainder.
3. Repeat until we get $r_{k+1} = 0$. Then $\gcd(a, b) = r_k$.

Example 3.40. Let us compute $\gcd(20, 78)$. We have $r_0 = 78$ and $r_1 = 20$. Then we compute

$$\begin{aligned} 78 &= 3 \cdot 20 + 18 & r_2 &= 18 \\ 20 &= 1 \cdot 18 + 2 & r_3 &= 2 \\ 18 &= 9 \cdot 2 + 0 & r_4 &= 0 \end{aligned}$$

and thus $\gcd(20, 78) = r_3 = 2$.

Example 3.41. Let's compute $\gcd(94012, 33396)$. We have $r_0 = 94012$ and $r_1 = 33396$.

$$\begin{aligned}
94012 &= 2 \cdot 33396 + 27220 & r_2 &= 27220 \\
33396 &= 1 \cdot 27220 + 6176 & r_3 &= 6176 \\
27220 &= 4 \cdot 6176 + 2516 & r_4 &= 2516 \\
6176 &= 2 \cdot 2516 + 1144 & r_6 &= 1144 \\
2516 &= 2 \cdot 1144 + 228 & r_7 &= 228 \\
1144 &= 5 \cdot 228 + 4 & r_8 &= 4 \\
228 &= 57 \cdot 4 + 0 & r_9 &= 0
\end{aligned}$$

so $\gcd(94012, 33396) = r_8 = 4$.

Fact 3.42 (Lamé)**.** *For any pair of natural numbers a, b, the Euclidean algorithm takes at most $\log_2(ab)$ steps to find (a, b).*

For any pair of natural numbers $a > b$, the Euclidean algorithm takes at most $5\log_{10}(b)$ steps to find (a, b).

So if we can find a large number L that fits our requirements—$p-1 \mid L$ and $q-1 \nmid L$—then we have a polynomial time algorithm for factoring N. But how do we find this?

Pollard says: first, we hope things work out well. And if they work out well, maybe $p-1$ doesn't have any large factors, but is the product of a bunch of small numbers. And if this is true, it will divide $n!$ for some small n. So we can compute $\gcd(a^{n!} - 1, N)$ for some small n values and hope that one of them gives us the answer we want.

In order to make this practical, we need to do a couple numeric reductions. The number $a^{n!}$ grows very large very quickly, even if we take $a = 2$; for instance, we know that $10! = 3628800$ and thus $2^{10!}$ has roughly a million digits in base 10, or 3.6 million bits. $2^{100!}$ has 10^{157} digits and can't be represented physically in the known universe. But since we just want the greatest common divisor with N, we can do all our calculations mod N, which will keep the numbers much smaller.

Further, if we already know $a^{n!}$, it's easy to compute $a^{(n+1)!} = (a^{n!})^{n+1}$. So if we luck out, the following algorithm will work:

Algorithm 3.8 (Pollard's $p-1$ algorithm)**.** Set $a_1 = 2$. Then for each $i > 1$:

1. Let $a_i = a_{i-1}^i$.

2. Compute $d = \gcd(a_i - 1, N)$.

3. If $1 < d < N$ then d is one of our prime factors of N, and we can stop.

4. Otherwise, repeat for the next i up.

How efficient is this? Computing $a^{n!} \mod N$ is $O(2n \log_2(n))$, which is roughly linear in n. If $p-1$ is made up of many small factors, then n will be small relative to $p-1$, and this will be efficient; but if p has large prime factors then n will be fairly large relative to $p-1$ and this is not efficient.

Example 3.43. Suppose we want to factor 1411. We'll take $a_1 = 2$, and then:

- $a_2 = 2^2 = 4$, and $\gcd(3, 1411) = 1$.
- $a_3 = 4^3 = 64$, and $\gcd(63, 1411) = \gcd(63, 25) = 1$.
- $a_4 = 64^4 = 16777216 \equiv 426 \mod 1411$, and $\gcd(425, 1411) = \gcd(425, 136) = \gcd(136, 17) = 17$.

Thus 17 is a non-trivial factor of 1411.

This particular problem can be guarded against: when Bob chooses his secret primes p and q, he can check that neither $p-1$ nor $q-1$ factors entirely into small primes. (Factoring in general is hard, but checking that there are large prime factors is not, because you just have to test-divide by all the "small" prime factors. Basically, Bob's checking step is always more efficient than Eve's Pollard $p-1$ algorithm.

3.6.2 Factoring with the Discrete Logarithm

RSA and ElGamal have many similarities, despite being based on different problems (integer factoring and the discrete logarithm, respectively). It turns out that this isn't an accident; both of these problems are examples of something called the *hidden subgroup problem* for finite abelian groups.

But we can also draw a more direct connection: if we can solve the discrete logarithm problem efficiently, then we can also factor large numbers efficiently.

Algorithm 3.9 (Factoring with Discrete Logarithms)**.** Suppose we want to factor a number N.

1. Choose a random $a < N$.

2. Compute $\gcd(a, N)$ with the Euclidean algorithm.

3. If this gcd is not 1, then we have a factor of N.

4. Otherwise, find $r = \log_a 1 \mod N$. (This is believed to be hard, but we're assuming we have a way to compute discrete logs).

5. If r is odd, start over with a different a.

6. If $a^{r/2} \equiv \pm 1 \mod N$, start over with a different a.

7. Otherwise, we have $a^{r/2}+1, a^{r/2}-1 \not\equiv \pm 1 \mod N$, and $(a^{r/2}+1)(a^{r/2}-1) = a^r - 1 \equiv 0 \mod N$. So $\gcd(N, a^{r/2}+1)$ and $\gcd(N, a^{r/2}-1)$ will be non-trivial factors of N.

Remark 3.44. There is a *quantum computer* algorithm that can solve the problem in step 4, known as Shor's Algorithm. Thus both ElGamal and RSA are vulnerable to a quantum computer. I'm hoping to talk about this at the end of the course.

Currently the largest number that has been successfully factored on a quantum computer is 21.

The other direction is more subtle and complicated, and doens't work as well as we'd like, but still works okay; if we could factor large numbers reliably, we would also be able to compute many discrete logarithms easily.

3.7 Elliptic Curves

One of the deepest, coolest, and most interesting fields of mathematics is the subject of elliptic curves. Fortunately for us, they're also widely used in cryptography today.

3.7.1 Groups and Fields

We need to introduce two fundamental ideas from algebra.

Groups

Definition 3.45. A *group* is a set G and a binary operation $\star : G \times G \to G$ with the properties that:

1. There is an *identity element* $e \in G$ such that $e \star g = g \star e = g$ for all $g \in G$;

2. For every $g \in G$, there is an *inverse element* g^{-1} such that $g \star g^{-1} = g^{-1} \star g = e$;

3. And the operation is *associative*, i.e. for every $f, g, h \in G$ we have $(f \star g) \star h = f \star (g \star h)$.

By convention we treat this operation as multiplication. A group always reflect some sort of symmetries; any time you have a collection of symmetries or a repeating pattern you can describe it as a group.

Example 3.46.
1. The integers with the operation of addition form a group $\mathbb{Z}$.
2. The non-zero rational numbers with the operation of multiplication form a group $\mathbb{Q}$.
3. The set of integers mod n, with the operation of addition mod n, forms a group $\mathbb{Z}/n\mathbb{Z}$.
4. The set of invertible $n \times n$ matrices with the operation of matrix multiplication form a group, called the *general linear group of degree* n or $GL(n)$.
5. The set of $n \times n$ matrices with determinant 1 with the operation of matrix multiplication form a group called the *special linear group of degree* n or $SL(n)$.
6. The set of rotations of a circle, with the operation of composition, form a group.
7. The set of permutations of a n-element set, with the operation of composition, forms a group called the *symmetric group on n letters* or S_n.

You'll note that in some of these groups, the operation is commutative: $a + b = b + a$. In others, like matrix multiplication or the permutations, the operation is not commutative. In general we don't assume groups are commutative; but many of the groups we're interested in in this class are.

Definition 3.47. Let $(G, \star)$ be a group. If the operation $\star$ is commutative, i.e. $g \star h = h \star g$ for all $g, h \in G$, then we say G is an *abelian* group.

If G is an abelian group, we often write $+$ for the operation and write $-g$ for the inverse of g and 0 for the identity. We *never* use addition notation if our group is not abelian.

We often want to fix an element g and consider the set of all the group elements we can get just from g.

Definition 3.48. Let $g \in G$. Then the set $\{g^n : n \in \mathbb{Z}\}$ is called the subgroup of G *generated by* g, often notated $\langle g \rangle$.

The size of this group is the *order of* g, and is the least positive integer $m = \operatorname{ord}_G(g)$ such that $g^m = e$.

If there is some $g \in G$ such that $G = \langle g \rangle$, we say that G is *cyclic* and that g is a *generator* for G.

This should sound very familiar from section 3.3.1. This is in fact a generalization; the idea of a generator here corresponds to the idea of a primitive root modulo m. We can also generalize one further fact:

Fact 3.49. *If G is a group of order n and $g \in G$ then* $\operatorname{ord}_G(g) | n$.

Fields A field is in essence a set in which we can do addition, multiplication, and division. Formally:

Definition 3.50. A *field* is a set K together with two operations $+$ and $\cdot$, such that

1. K is an abelian group under the operation $+$;
2. The set $K \setminus \{0\}$ of non-zero elements of K is an abelian group under $\cdot$;
3. and we have the distributive law $k(x + y) = kx + ky$.

Example 3.51.
1. The real numbers, the rational numbers, and the complex numbers are all fields.
2. The integers are *not* a field, since they don't form a group under multiplication: there are no inverses. The requirement that we have a *group* under multiplication is what allows division.

The most important example for us is $\mathbb{Z}/p\mathbb{Z}$. This clearly forms a group under addition. Further, we know that every non-zero element is invertible mod p, so the non-zero elements form a group under multiplication. Thus $\mathbb{Z}/p\mathbb{Z}$ is a field. It is in fact the only field of order p, and when we're thinking of it as a field we often denote it $\mathbb{F}_p$.

In contrast, if m is composite, then $\mathbb{Z}/m\mathbb{Z}$ is not a field, since any factor of m will not be invertible mod m.

Typically when we're looking for solutions to some equation, we need to specify the field we're working in. We know this already: The equation $x^2 - 2 = 0$ has solutions in the reals but not in the rationals, and $x^2 + 1 = 0$ has solutions in the complex numbers but not in the reals.

For our cryptographic applications, we typically want to be working in the field $\mathbb{F}_p$ for some large prime p. But the important idea here is that most of our algebra works the same in any field. We can add, subtract, multiply, and divide the same way regardless.

3.7.2 Elliptic Curves

Definition 3.52. An elliptic curve over a field K is a smooth projective curve over K of genus 1, together with a point defined over K.

That...doesn't mean very much. Let's try again.

Definition 3.53. An *elliptic curve* over a field K is given by an equation of the form

$$y^2 = x^3 + Ax + B \tag{4}$$

where $A, B \in K$, provided that the discriminant $\Delta = 4A^3 + 27B^2 \neq 0$. The set of solutions to this equation with coordinates in K is the set of points of the elliptic curve, and is denoted $E(K)$.

We say a curve written this way is in *Weierstrass Form*.

The condition on the discriminant guarantees that the cubic does not have any repeated roots; if we factor $x^3 + Ax + B = (x - \alpha_1)(x - \alpha_2)(x - \alpha_3)$, then $\Delta = 4A^3 + 27B^2 \neq 0$ if and only if $\alpha_1, \alpha_2, \alpha_3$ are distinct. This turns out to be important for the group law we wish to build.

Remark 3.54. An elliptic curve is *not* an ellipse. The name comes from the relationship to elliptic integrals, which are the integrals you need to compute the circumference of an ellipse.

Over the complex numbers, an elliptic curve forms a torus. Over the reals, we get a characteristic pinched curve or circle-and-wiggle curve.

Elliptic curves start off as an example of a basic number theory question: when does a given equation have solutions in the rational numbers? But elliptic curves have a very nice property that makes this question in some ways much easier: the group law.

Bezout's Theorem The group law depends on a result from algebraic geometry known as Bezout's Theorem. We say that a planar curve is of *degree d* if it is defined by a polynomial whose highest term is degree d.

Theorem 3.55 (Bezout's Theorem). *Suppose C_1 is a curve of degree d and C_2 is a curve of degree e. Then there are exactly $e \cdot d$ points in the intersection $C_1 \cap C_2$, up to some technical conditions.*

It's actually really easy to come up with counterexamples to Bezout's Theorem without the technical conditions. One is that Bezout's Theorem holds over the complex numbers but not over the reals (since, say, the intersection of $x = -1$ and $x = y^2$ has no real solutions but two complex solutions). This condition won't actually be important for the application we care about.

There are two other technicalities that are more important. The first is that when we're counting the number of intersections, some of them count "more than once". This corresponds to the idea that $(x-1)^5$ has one root "five times", and thus we can say that any degree-d polynomial has exactly d roots over the complex numbers.

We can visualize some of this by thinking of the intersection of a circle and a line. If the line doesn't hit the circle, then they have no real intersections (but two complex intersections, since the line has degree 1 and the circle has degree 2). If the line goes through the circle, then there are two actual real intersections we can see. And if the line is *tangent* to the circle, there is one intersection but it counts twice.

The second, and most interesting, is the introduction of points "at infinity". The lines $y = 1$ and $y = 2$ are each degree 1, so should intersect at exactly one point. Obviously they have no points in common, though. We introduce "points at infinity" so that any pair of parallel lines has exactly one intersection. (The formalization of this idea is called "projective geometry").

Definition 3.56. We can define an equivalence relation on $\mathbb{R}^3 \setminus \{(0,0,0)\}$ by saying $(a,b,c) \sim (x,y,z)$ if there is a real number r such that $r(a,b,c) = (x,y,z)$. Thus, two points are equivalent if they are on the same line through the origin.

We define the *(real) projective plane* $\mathbb{P}_2(\mathbb{R})$ to be the set of these equivalence classes. Another way of looking at this is that $\mathbb{P}_2(\mathbb{R})$ is the set of lines through the origin in $\mathbb{R}^3$.

Why do we call this a plane? Well, if we look at all the points such that $z \neq 0$, we can normalize them so that $z = 1$. Then we have a collection of points $\{(x,y,1) : x, y \in \mathbb{R}\}$, which forms a plane. But we're also left with some "extra" points, where $z = 0$. These give us our "points at infinity".

Why do we call this a "projective" plane? The plane defined by $z = 1$ is a plane one unit above the origin. So what we're doing is we're taking every point and projecting it through the origin onto this plane. Our extra points, with $z = 0$, don't actually project onto this plane when we draw a line through the origin. But we can imagine them projecting on to the "edge" of the plane at infinity.

(The weirdest thing about this, perhaps, is that different infinite directions give different points, but directly opposite directions give the same point; the line through the origin and

$(1, 0)$ is the same as the line through the origin and $(0, 1)$. Thus the "boundary" of the plane doubles up on itself.

Geometry and the Group Law How does this let us find points on an elliptic curve? The elliptic curve has degree 3, so any line should intersect it in exactly three points. Thus if we have any two points on an elliptic curve, we can draw a line through the two points and find a third point of intersection.

This won't get us terribly far on its own, though—we just get one new point. In order to get us out of the rut we could get stuck in, we actually reflect our new point across the line $y = 0$—because every y in our defining equation is squared, then whenever (x, y) is a point on the curve, so is the point $(x, -y)$.

Definition 3.57. Let P, Q be two points on an elliptic curve. We can draw a line through them and find a third point on the elliptic curve, which we'll call R. Let R' be the reflection of R across the x-axis. Then we define $P \oplus Q = R'$.

Remark 3.58. This is very specifically *not* the result we'd get by adding the coordinates of P and Q; doing that will probably not give another point on the elliptic curve.

There are a couple special cases we need to deal with. The first is when our line has intersection with multiplicity. In particular, the only way this can happen is if the line is tangent to the elliptic curve, in which case it intersects it with multiplicity two at the point of tangency. So if the line through P and Q is tangent to the curve at P, then our point R from this algorithm is just P. And if we want to compute $P \oplus P$ then we draw the line tangent to the curve at P and find the third point of intersection (and then reflect across the x-axis).

The other special case is to reintroduce the projectivity. Recall that Bezout's theorem only holds if you allow points at infinity. In particular, we say that every elliptic curve contains a point at infinity which is intersected by every vertical line; we call this point $\mathcal{O}$.

Remark 3.59. We can actually recast our "addition" law as drawing a line through P and Q and finding the third point of intersection R; and then drawing a line through R and $\mathcal{O}$ and finding the third point of intersection R', and then defining $P \oplus Q = R'$. This is the same definition as earlier, slightly generalized.

How does the $\oplus$ operation work with $\mathcal{O}$? Every line through $\mathcal{O}$ and another point is vertical. So we can see that if P is any point, then the line through $\mathcal{O}$ and P also intersects the reflection of P across the x-axis; reflecting this back across the x-axis gives P, so $\mathcal{O} \oplus P = P$. Thus $\mathcal{O}$ is an identity for the $\oplus$ operation.

(You might want to check here that $\mathcal{O} \oplus \mathcal{O} = \mathcal{O}$. This is in fact true, because a line can intersect $\mathcal{O}$ with multiplicity 3. But proving that is far beyond the scope of this discussion, so we'll just take it as a definition for now).

We'd like to show that this operation gives us an abelian group. The operation is clearly commutative, since the line through P and Q is the same as the line through Q and P. Associativity is straightforward, but incredibly tedious. And we already have an identity. So we just need to show that we have inverses.

Let P be a point, and let Q be the reflection of P across the x-axis. Then the line through P and Q is vertical, and the third point of intersection is $\mathcal{O}$. The line through $\mathcal{O}$ and $\mathcal{O}$ has its third point of intersection in $\mathcal{O}$, so we see that $P \oplus Q = \mathcal{O}$, and by definition Q is an inverse of P. Thus in general we notate the reflection of P across the x-axis by $-P$.

Proposition 3.60. *Let E be an elliptic curve defined over a field K. Then the set of points $E(K)$ forms a group under the operation $\oplus$ defined above.*

Definition 3.61. We write $P - Q$ for $P \oplus (-Q)$ and we write $nP = P \oplus \cdots \oplus P$.

This operation can be characterized in another, possibly more intuitive way.

Fact 3.62. *If P, Q, R are points of an elliptic curve all on the same line, then $P \oplus Q \oplus R = \mathcal{O}$.*

Thus if P, Q, R are all on the same line, then $P \oplus Q = -R$.

3.7.3 Elliptic Curves over $\mathbb{Q}$

Let's first discuss elliptic curves over the rational numbers. These are not per se useful for cryptography, but they have two nice properties. One is that they're in many ways more theoretically rich and interesting; the other is that they are far easier to visualize.

It's a very difficult question in general to find all the points on an elliptic curve—or, indeed, any of them. It's not trivial to figure out whether an elliptic curve *has* any points over $\mathbb{Q}$ other than $\mathcal{O}$. The primary use of the group law is to constrain how many points an elliptic curve can have, and then to help us find them. If we start with one point, we can use the group law to repeatedly add it to itself to generate more.

Fact 3.63. *If $E(\mathbb{Q})$ is finite, then either $E(\mathbb{Q}) = \mathbb{Z}/n\mathbb{Z}$ for $n \in \{1, 2, \ldots, 9, 10, 12\}$, or $E(\mathbb{Q}) = \mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$ for $n \in \{2, 4, 6, 8\}$.*

Conjecture 3.64. *Exactly 50% of elliptic curves have an infinite number of points.*

However, if we do have a point, we can do arithmetic with it straightforwardly. Suppose we want to add the points (x_1, y_1) and (x_2, y_2). Then the line including both of these points will have equation

$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1. \tag{5}$$

If we substitute this into our elliptic curve equation, we should get three solutions for x; two of them will be x_1 and x_2, and the third will be x_3, the x-coordinate we're looking for. We plug x_3 back into the equation for the line to get $-y_3$, and multiply the y-coordinate by -1 to get our final point $(x_1, y_1) \oplus (x_2, y_2)$.

Example 3.65. Let's consider the elliptic curve $E : y^2 = x^3 - 15x + 18$. We can calculate the discriminant $\Delta = 4(-15)^3 + 27(18)^2 = -4752 \neq 0$, so this is in fact an elliptic curve.

We can check that $P = (7, 16)$ and $Q = (1, 2)$ are both in $E(\mathbb{Q})$. Let's compute $P \oplus Q$. The line through these points is given by

$$y = \frac{16 - 2}{7 - 1}(x - 1) + 2 = \frac{7}{3}(x - 1) + 2 = \frac{7}{3}x - \frac{1}{3}.$$

Substituting this into the equation for the elliptic curve gives

$$\begin{aligned} (7x/3 - 1/3)^2 &= x^3 - 15x + 18 \\ 49x^2/9 - 14x/9 + 1/9 &= x^3 - 15x + 18 \\ 0 &= x^3 - \frac{49}{9}x^2 - \frac{121}{9}x + \frac{161}{9}. \end{aligned}$$

This is a cubic equation, which is kind of annoying to solve. (There is a cubic formula analogue to the quadratic formula, but it's considerably more complex). However, we have an advantage since we already know two of the roots. This gives us two easy ways out.

The less easy way is to do polynomial long division. We can divide this polynomial by $(x - 1)$ and then by $(x - 7)$ to find the third term; we get

$$x^3 - \frac{49}{9}x^2 - \frac{121}{9}x + \frac{161}{9} = (x - 7)(x - 1)(x + 23/9)$$

so the third root is $-23/9$.

But there's an even easier way. We know that our polynomial is equal to $(x - 7)(x - 1)(x - x_3)$; we can check then that the coefficient of x^2 must be equal to $-7 - 1 - x_3$. Thus we set up

$$\begin{aligned} -49/9 &= -7 - 1 - x_3 \\ -49/9 + 8 &= -x_3 \\ -23/9 &= x_3. \end{aligned}$$

We know the x-coordinate of the third point on the line, so now we need the y-coordinate. We can get this easily by substituting back into the equation for our line, and we get

$$y = \frac{7}{3}(-23/9) - \frac{1}{3} = \frac{-161}{27} - \frac{1}{3} = \frac{-170}{27}.$$

Finally we reflect across the x-axis, to get $P \oplus Q = (-23/9, 170/27)$.

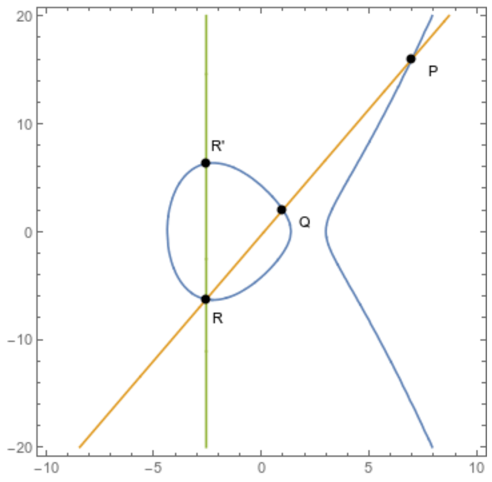

Figure 3.1: Calculating $P \oplus Q$

Example 3.66. Now let's compute $Q \oplus Q$. Recall that when we have a point occur twice, that means we want the tangent line at that point. To find the slope we use calculus. We know that

$$\begin{aligned} y^2 &= x^3 - 15x + 18 \\ 2yy' &= 3x^2 - 15 \\ 2 \cdot 2 \cdot y' &= 3 \cdot (1)^2 - 15 \\ y' &= \frac{-12}{4} = -3. \end{aligned}$$

Thus the tangent line at Q is given by

$$y = -3(x-1) + 2 = -3x + 5.$$

Substituting this into our curve gives

$$\begin{aligned} (5-3x)^2 &= x^3 - 15x + 18 \\ 25 - 30x + 9x^2 &= x^3 - 15x + 18 \\ 0 &= x^3 - 9x^2 + 15x - 7. \end{aligned}$$

Using the same trick as before, we know that $x^3 - 9x^2 + 15x - 7 = (x-1)(x-1)(x-x_3)$ so we have $-1 - 1 - x_3 = -9x^2$, giving us $x_3 = 7$.

To find the y-coordinate of the intersection, we substitute this into our linear equation, and get $y = -21 + 5 = -16$. Reflecting this across the x-axis, we get $Q \oplus Q = (7, 16) = P$.

Thus we've also seen that $3Q = (-23/9, 170/27)$.

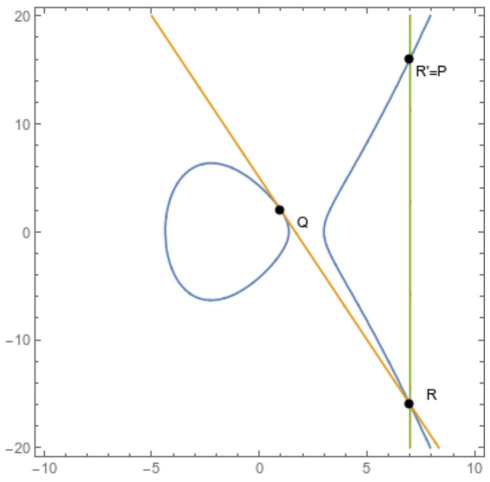

Figure 3.2: Calculating $2Q$

3.8 Elliptic Curve Cryptography

3.8.1 Elliptic Curves over a Finite Field

For the purposes of cryptography, we want to consider an elliptic curve defined over a finite field $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ for p a prime. Given a specific curve, we can find all of the points on it by exhaustive search.

Example 3.67. Let $E : y^2 = x^3 + 3x + 8$ be an elliptic curve over $\mathbb{F}_{13}$. (We check that $4 \cdot 3^3 + 27 \cdot 8^2 = 1836 \equiv 3 \not\equiv 0 \mod 13$ so this is an elliptic curve.

If we want to find all the points on this elliptic curve, we can plug in the values $0, 1, 2, \ldots, 12$ for x and then see if the equation $y^2 \equiv a \mod 13$ has solutions (for each number, it will have either zero or two).

We start by making a list of all the squares mod 13. We see that

$1^2 \equiv 1$	$2^2 \equiv 4$	$3^3 \equiv 9$	$4^3 \equiv 3$	$5^2 \equiv 12$	$6^2 \equiv 10$
$7^2 \equiv 10$	$8^2 \equiv 12$	$9^2 \equiv 3$	$10^2 \equiv 9$	$11^2 \equiv 4$	$12^2 \equiv 1$

(You might notice that the second row is just the first row backwards. This is because $(-a)^2 \equiv a^2 \mod p$. Thus $y^2 \equiv a \mod 13$ has a solution if and only if $a \in \{1, 3, 4, 9, 10, 12\}$.

So now we check values for x. If $x = 0$ then we have $y^2 \equiv 8$, which has no solutions. If $x = 1$ then we have $y^2 \equiv 12$, which has the solutions $y \equiv 5$ and $y \equiv 8$. Continuing we get the list:

$$E(\mathbb{F}_{13}) = \{\mathcal{O}, (1,5), (1,8), (2,3), (2,10), (9,6), (9,7), (12,2), (12,11)\}.$$

Thus we see $E(\mathbb{F}_{13})$ has nine points.

How do we do our point addition on these curves? It's really hard to draw pictures of these things that look reasonable, since it's just a scatter of points (see figure 3.5 for an example of a picture here). But we can still write down the same *equations* we always would.

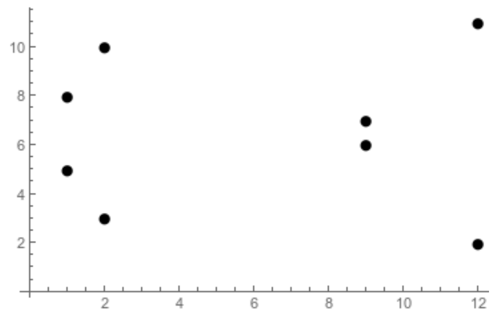

Figure 3.3: The curve $E : y^2 = x^3 + 3x + 8$ over $\mathbb{F}_{13}$

Example 3.68. Let's use the same elliptic curve as above, and let's calculate $(1,5) \oplus (9,6)$. Our line has the equation

$$y = \frac{6-5}{9-1}(x-1) + 5.$$

We need to figure out what 1/8 is—that is, the inverse of 8 modulo 13. A little experimentation gives us the $8 \cdot 5 = 40 \equiv 1 \mod 13$ so our equation becomes

$$y = 5(x-1) + 5 = 5x.$$

(We check that both our points are on this line; we see that $5 \cdot 1 = 5 \equiv 5$, and $9 \cdot 5 = 45 \equiv 6$).

Plugging this into our original equation gives

$$\begin{aligned}
(5x)^2 &= x^3 + 3x + 8 \\
25x^2 &= x^3 + 3x + 8 \\
0 &= x^3 - 25x^2 + 3x + 8 \\
&\equiv x^3 + x^2 + 3x + 8.
\end{aligned}$$

This seems like it might be painful to solve, but we have effectively three approaches. The first is simply trial and error; there are only thirteen possibilities, so we can just try them all. (This works well as long as p is small).

The second is polynomial long division. We already know two roots of this polynomial: 1 and 9. (We can check that both of these are roots to make sure we haven't screwed anything up). So we can long divide by $(x-1)$ and then by $(x-9)$; we see that

$$x^3 + x^2 + 3x + 8 = (x-1)(x^2+2x+5) = (x-1)(x-9)(x-2).$$

But the third approach extends this to be easier still. We know that our polynomial will be $(x-1)(x-9)(x-x_3)$ for some x_3. Thus in particular, we can see that $-1-9-x_3$ will be the coefficient of x^2. Thus we have $-1-9-x_3 \equiv 1 \mod 13$ and so $-x_3 \equiv 11$, so $x_3 \equiv 2$.

Plugging $x = 2$ back intou our line equation gives $y = 10$, so the third point on the line through $(1,5)$ and $(9,6)$ is $(2,10)$. (We check that this point is actually on the curve; indeed, it is).

Our last step is to reflect this point vertically, to get $(2,-10) \equiv (2,3)$. Thus $(1,5) \oplus (9,6) = (2,3)$.

We can attempt to draw a picture here, but it's not super helpful. here's a picture of the line through P and Q, and then a picture of that line overlaid over E:

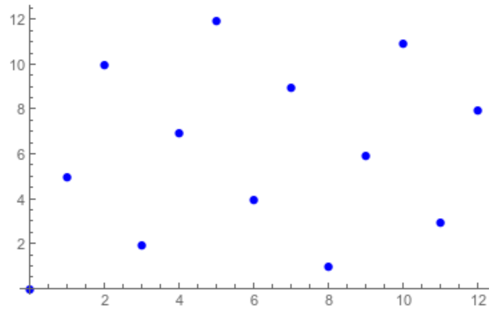

Figure 3.4: The $y = 5x$ through $(1,5)$ and $(9,6)$ over $\mathbb{F}_{13}$

As you can see, it's somewhat challenging to figure out what's going on here even already knowing the answer! This is why we turn our questions of geometry over finite fields into questions of algebra.

Example 3.69. Let's do another example. This time we'll calculate $(12,2) \oplus (12,2)$.

Since we're adding a point to itself, we can't just find the equation of the line going through both points. Instead we need to find the tangent line. It's not necessarily clear exactly what this should mean in modular arithmetic—there certainly isn't a curve in the

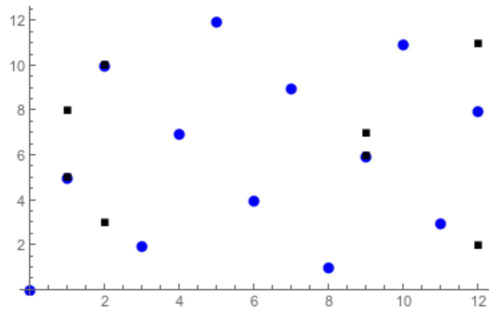

Figure 3.5: The curve $E : y^2 = x^3 + 3x + 8$ (black squares) and the line $y = 5x$ (blue circles)

picture—so we'll just fall back on what the answer "should" be from regular calculus. (I could make this rigorous. I won't). So we calculate

$$\begin{aligned}
y^2 &\equiv x^3 + 3x + 8 \\
2yy' &\equiv 3x^2 + 3 \\
2 \cdot 2 \cdot y' &\equiv 3(-1)^2 + 3 \\
4y' &\equiv 6 \\
2y' &\equiv 3 \\
y' &\equiv 8.
\end{aligned}$$

Thus our line is $y \equiv 8(x - 12) + 2$ or $y \equiv 8x + 10$. Again we can plug this into our elliptic curve:

$$\begin{aligned}
(8x + 10)^2 &\equiv x^3 + 3x + 8 \\
64x^2 + 160x + 100 &\equiv x^3 + 3x + 8 \\
-x^2 + 4x + 9 &\equiv x^3 + 3x + 8 \\
0 &\equiv x^3 + x^2 - x - 1.
\end{aligned}$$

As before we know that $x^3+x^2-x-1 \equiv (x-12)(x-12)(x-x_3)$ so we have $-12-12-x_3 \equiv 1$, or $x_3 \equiv 1$. Plugging this into the line equation gives $y \equiv 8 + 10 \equiv 5$, so the third point on this line is $(1, 5)$ (which is in fact on $E(\mathbb{F}_{13})$). We invert the y-coordinate, so we get $(12, 2) \oplus (12, 2) = (1, 8)$.

3.8.2 The group law by formula

Notice that while we could—and did—work though every step of ellpitic curve addition in detail, most of the work we did is brute algebra, and can be automated into formulas.

Proposition 3.70. *Let $E : y^2 = x^3 + Ax + B$ be an elliptic curve over a field K, and let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be points on $E(K)$. Then:*

1. *If $y_1 = -y_2$ (in K), then $P \oplus Q = \mathcal{O}$.*
2. *If $P = Q$, then define $\lambda = \frac{3x_1^2+A}{2y_1}$. Set*

$$x_3 = \lambda^2 - x_1 - x_2 \qquad y_3 = \lambda(x_1 - x_3) - y_1.$$

   Then $P \oplus Q = (x_3, y_3)$.

3. *If $P \neq Q$, then define $\lambda = \frac{y_2-y_1}{x_2-x_1}$. Then as before, set*

$$x_3 = \lambda^2 - x_1 - x_2 \qquad y_3 = \lambda(x_1 - x_3) - y_1.$$

   Then $P \oplus Q = (x_3, y_3)$.

Proof. This all follows from the sort of algebraic arguments we just made. We take λ to be the slope of the line through P and Q—the formula if $P = Q$ comes from setting $2yy' = 3x^2 + A$ so that $y' = \frac{3x^2+A}{2y}$.

The formula from x_3 comes from the observation that the coefficient of x^2 in the cubic we're solving is always $-\lambda^2$, so we have $-x_1 - x_2 - x_3 = -\lambda^2$ or $x_3 = \lambda^2 - x_1 - x_2$. The formula for y_3 comes from plugging x_3 into the equation of the line and then multiplying by -1. □

Example 3.71. Let's do one more addition on our elliptic curve $E : y^2 = x^3 + 3x + 8$ over $\mathbb{F}_{13}$. Let's compute $(2, 3) \oplus (2, 3)$. We observe that this is a repeated point, so we have

$$\lambda = \frac{3x_1^2 + A}{2y_1} = \frac{3 \cdot 2^2 + 3}{2 \cdot 3} = \frac{15}{6} = \frac{5}{2} = 9.$$

Then we have

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 = 9^2 - 2 - 2 = 60 = 12 \\ y_3 &= \lambda(x_1 - x_3) - y_1 = 9(2 - 12) - 3 = -93 = 11 \end{aligned}$$

so $(2, 3) \oplus (2, 3) = (12, 11)$. (This is in fact a point on the curve E, which is good).

Example 3.72. In section 3.7.3 we worked with the elliptic curve $E : y^2 = x^3 - 15x + 18$ over the field $\mathbb{Q}$, with $Q = (1, 2), Q \oplus Q = P = (7, 16)$, and $Q \oplus P = 3\mathbb{Q} = S = (-23/9, 170/27)$. Now let's compute $Q \oplus S$.

Our two points are distinct, so we have

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} = \frac{170/27 - 2}{-23/9 - 1} = \frac{-29}{24}.$$

Then we have

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 = \frac{29^2}{24^2} - 1 - \frac{-23}{9} = \frac{193}{64} \\ y_3 &= \lambda(x_1 - x_3) - y_1 = \frac{-29}{24}\left(1 - \frac{193}{64}\right) - 2 = \frac{223}{512}. \end{aligned}$$

Thus $Q \oplus S = (193/64, 223/512)$.

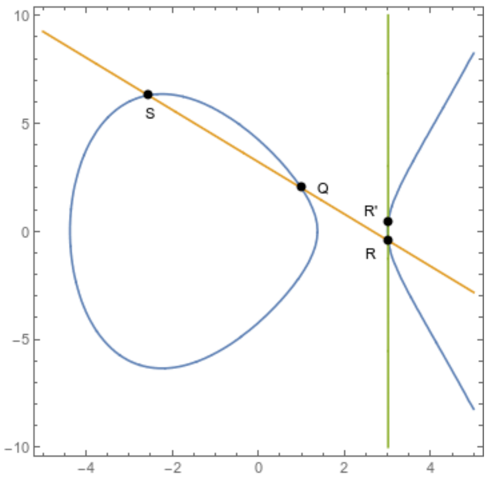

Figure 3.6: Calculating $2Q$

Theorem 3.73 (Hasse)**.** *Let E be an elliptic curve over $\mathbb{F}_p$. Then $|\#E(\mathbb{F}_p) - (p+1)| < 2\sqrt{p}$. Thus an elliptic curve over $\mathbb{F}_p$ has about $p+1$ points on it.*

Remark 3.74. The error term $p + 1 - \#E(\mathbb{F}_p) = t_p$ is called the *trace of Frobenius*, and is the trace of a 2×2 matrix acting on a vector space associated to $E/\mathbb{F}_p$. The details of this are technical, and completely irrelevant to our purposes, but are very important to my PhD research on special values of L-functions (and the Birch and Swinnerton-Dyer Conjecture).

3.8.3 Elliptic Curve Discrete Logarithms

In order to use elliptic curves in cryptography, we need a problem that is difficult in one direction, and easy in the other—an analogue of the discrete logarithm problem. Fortunately, exactly such a problem exists.

Definition 3.75. Let E be an elliptic curve over the finite field $\mathbb{F}_p$ and let $P, Q \in E(\mathbb{F}_p)$ be points on the curve. The *elliptic curve discrete logarithm problem* is the problem of finding an integer n such that $Q = nP$ (where the operation is the group law on the elliptic curve).

By analogy, we denote this integer by $n = \log_P(Q)$ and call n the elliptic discrete logarithm of Q with respect to P (on the curve $E/\mathbb{F}_p$) .

Example 3.76. For $E : y^2 = x^3 + 3x + 8$, what is $log_{(2,3)}(1, 8)$? That is, for what n do we have $n(2, 3) = (1, 8)$?

We already saw that $2(2, 3) = (12, 11)$. We can then compute that

$$\begin{aligned} 3(2,3) &= (12,11) \oplus (2,3) = (9,7) \\ 4(2,3) &= (9,7) \oplus (2,3) = (1,5) \\ 5(2,3) &= (1,5) + (2,3) = (1,8) \end{aligned}$$

so $5(2, 3) = (1, 8)$ and we have $\log_{(2,3)}(1, 8) = 5$.

Like the regular discrete logarithm problem, it's somewhat time-consuming and expensive to compute an elliptic curve discrete logarithm. But also like the regular discrete logarithm problem, it's fairly efficient to do the opposite.

Algorithm 3.10 (Double and Add)**.** Suppose we want to compute nP for some point $P \in E(\mathbb{F}_p)$. Let $k = \log_2(n)$. Then we can:

1. Compute $2^k P$ for $2^k \leq a$. That is, compute $g, g^2, g^4, g^8, \ldots, g^{2^k}$. We can do this by repeated squaring, without computing intermediate powers.
2. Now express n in binary. That is, write $n = c_0 + c_1 \cdot 2 + c_2 \cdot 2^2 + \cdots + c_k 2^k$, where $c_i \in \{0, 1\}$.
3. Now we can compute

   $$nP = (c_0 + c_1 \cdot 2 + c_2 \cdot 2^2 + \cdots + c_k 2^k)P = c_0 P \oplus c_1 2P \oplus c_2 4P \oplus \cdots \oplus c_k 2^k P$$

   But we already know $2^i P$ for each i, and the c_i are all either 0 or 1 so don't involve any computation. So we only have to do about $2k$ elliptic curve additions.

This should look familiar, since it's exactly the fast exponentiation algorithm we've already seen.

Remark 3.77. We could actually get an additional (minor) speedup by exploiting the fact that elliptic curve subtraction is just as easy as addition—this is *not* analogous to the mod-p multiplication case.

Example 3.78. For $E : y^2 = x^3 + 3x + 8$, let's compute $9(2,3)$. We already know that $2(2,3) = (12,11)$ and $4(2,3) = (1,5)$. Then we have $8(2,3) = (1,5) \oplus (1,5) = (2,10)$. Thus $9(2,3) = (2,3) \oplus (2,10) = \mathcal{O}$.

Remark 3.79. Could we have known that $9(2,3) = \mathcal{O}$ without doing any calculations? Yes!

We know that E had eight "normal" points on it, plus the identity, for a total of nine elements in the additive group. Thus $9P = \mathcal{O}$ for any $P \in E(\mathbb{F}_{13})$.

We can adapt the baby step-giant step algorithm to attack the elliptic curve discrete logarithm problem, as we did for the regular discrete logarithm problem.

Fact 3.80. *Optimal (known) fast elliptic curve multiplication takes about* $3k/2+1$ *operations in the worst case, and* $4k/3+1$ *steps on the average, where* $k = \log_2(n)$. *(Algorithm 3.10 takes about* $2k$ *and* $3k/2$ *respectively, instead).*

Optimal (known) discrete logarithm solving takes about $\sqrt{p}$ *steps, where* p *is the order of the field that* E *is defined over.*

3.8.4 Cryptographic Algorithms

We can, effectively, implement all of our discrete logarithm-based cryptographic algorithms with elliptic curve discrete logarithms instead.

Algorithm 3.11 (Elliptic Curve Diffie-Hellman)**.** Alice and Bob wish to exchange a key. They follow the following steps:

1. A public party chooses a large prime p, and an elliptic curve E over $\mathbb{F}_p$, and a point $P \in E(\mathbb{F}_p)$.

2. Alice chooses a secret integer n_A, and Bob chooses a secret integer n_B. Neither party reveals this integer to anyone.

3. Alice computes $Q_A = n_A P$ and Bob computes $Q_B = n_B P$. They (publicly) exchange these values with each other.

4. Now Alice computes $n_A Q_B$ and Bob computes $n_B Q_A$.

5. $n_A Q_B = n_A n_B P = n_B n_A P = n_B Q_a$, so they now have a shared key.

Algorithm 3.12 (Elliptic Curve ElGamal)**.** First Alice generates a private key and a public key.

1. Choose a large prime number p, an elliptic curve E over $\mathbb{F}_p$, and a point $P \in E(\mathbb{F}_p)$ of large order. This is generally done by a large trusted party.
2. Alice chooses a private key n_A.
3. Alice computes and publishes a public key $Q_A = n_A P \in E(\mathbb{F}_p)$.

Now suppose Bob wishes to send Alice a a message encoded as a point $M \in E(\mathbb{F}_p)$.

1. Bob generates a random ephemeral key k.
2. Bob computes $C_1 = kP \in E(\mathbb{F}_p), C_2 = M + kQ_A \in E(\mathbb{F}_p)$. Bob transmits the pair of points (C_1, C_2) to Alice.

Alice decrypts the message using her private key n_A. She computes $C_2 - n_A C_1 \in E(\mathbb{F}_p)$. We see that this is

$$C_2 - n_a C_1 = M + kQ_A - n_A kP = M + kn_A P - n_A kP = M.$$

Remark 3.81. Elliptic curve cryptography introduces one layer of added inefficiency: a single point contains about $\log_2(p)$ bits of information, since an elliptic curve over $\mathbb{F}_p$ has about p points. But since we need to transmit two numbers mod p to convey one point, we have to send $2\log_2(p)$ bits of information!

There are various hacks to get around this problem. Most of them involve the fact that if we know the x-coordinate of a point on E, then we know the y-coordinate up to a change of sign. So there are schemes involving only caring about the x coordinate, or involving transmitting the x-coordinate and a parity bit to specify which of the two possible y-coordinates you had chosen, rather than transmitting the entire number.

Another issue with elliptic curve cryptography is the choice of curve and point. It is computationally expensive to ensure that a curve does not have any hidden weaknesses, so curves are generally chosen once by a respected standards body. In practice in the USA, they are often chosen by NIST.

In 2013, the New York Times revealed that some of the standard curves chosen by NIST were chosen due to influence by the NSA, which had introduced a secret weakness into the chosen curves, which allowed it to crack encryption based on those curves.

However, elliptic curve cryptography has a major advantage over algorithms based on modular arithmetic: the General Number Field Sieve. This algorithm, which we discussed in 3.6, is the most efficient known algorithm for breaking RSA, and can also be adapted to attack vanilla ElGamal and most similar cryptosystems based on modular arithmetic. But nothing similar is known in the case of elliptic curves. Thus elliptic curve cryptography provides substantially greater strength for the same size key.

Under current knowledge and assumptions, we have the following table, explaining what key lengths provide equivalent levels of security for symmetric algorithms, RSA, and ECC.

Symmetric Key Size	RSA Key Size	ECC Key Size
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Thus modern RSA often uses 2048 bits. AES, which is one of the most common symmetric algorithms, uses 128, 192, or 256 bits instead. Modern ECC uses 224 or 256 bits. Thus ECC keys can be much shorter, and in turn computations with them can be much faster.