Math 173A

Liam Hardiman

July 20, 2022

Abstract

I'm writing these lecture notes for UC Irvine's Math 173A course, taught in the summer of 2022. This is a five-ish week course where I plan to get through the first three chapters of Hoffstein, Pipher and Silverman's book [1]. The class structure consists of a two hour lecture followed by a one hour discussion section three days a week. I'm aiming to get through two sections of the book per lecture with a midterm after chapter 2.

Contents

1 An Introduction to Cryptography

1.1 Simple Substitution Ciphers

One of history's oldest examples of encrypting messages is the *shift cipher*, sometimes called the *Caesar cipher* after Julius Caesar, who allegedly used it to encrypt the orders he'd send to his troops. To encrypt a message, simply shift each letter of the plaintext forward in the alphabet by three, wrapping around if the shifted letter goes past `Z`. For example, if the key[1] is `3` and our plaintext is `hello, world`, then we have the following ciphertext.

`hello world` $\mapsto$ `KHOOR ZRUOG`

Conversely, if we know the key is `3` and we're given the ciphertext `ZHGQH VGDB`, then we simply shift backwards by 3 to obtain the plaintext.

`ZHGQH VGDB` $\mapsto$ `wedne sday`

One advantage to the shift cipher is that it's really easy to encrypt and decrypt messages if the key is known. The main disadvantage is that it's only slightly challenging (more annoying than challenging) for an adversary to decrypt messages even if they don't know the key. If we use the English alphabet, then there are only 26 possible keys and it doesn't take too long to try them all (a few minutes by hand, a fraction of a second even with bad code). This trial and error method of trying all possible keys, sometimes called *brute forcing*, works because it's pretty unlikely that decrypting with two different keys will yield two plaintexts that are both readable. For example, suppose we happen upon the following ciphertext

`XPPEE ZXZCC ZH.`

If we suspect that this ciphertext came from a shift cipher, we can just try all possible un-shifts to get the following possible plaintexts.

[1] We won't rigorously define what "plaintext", "ciphertext" or "key" mean. You can think of the plaintext as being being the human-readable or usable message (maybe consisting of letters or a number) and the ciphertext as being some unreadable sequence of letters or numbers. Then you can think of the key as being some piece of information that tells you how to convert between plain- and ciphertext.

key	plaintext	key	plaintext
1	woodd ywybb yg	14	jbbqq ljloo lt
2	vnncc xvxaa xf	15	iaapp kiknn ks
3	ummbb wuwzz we	16	hzzoo jhjmm jr
4	tllaa vtvyy vd	17	gyynn igill iq
5	skkzz usuxx uc	18	fxxmm hfhkk hp
6	rjjyy trtww tb	19	ewwll gegjj go
7	qiixx sqsvv sa	20	dvvkk fdfii fn
8	phhww rpruu rz	21	cuujj ecehh em
9	oggvv qoqtt qy	22	bttii dbdgg dl
10	nffuu pnpss px	23	asshh cacff ck
11	meett omorr ow	24	zrrgg bzbee bj
12	lddss nlnqq nv	25	yqqff ayadd ai
13	kccrr mkmpp mu		

The only plaintext here that's even remotely readable is `meett omorr ow`, corresponding to a key of `11`. This process of decrypting a ciphertext without knowing the key in advance is called *cryptanalysis*.

Notice that with a shift cipher, each instance of `a` encrypts to the same character, and so on. In this setting, once we know what one character maps to, then we know what all the other characters map to as well. E.g. if we know that `m` maps to `X`, then we know that the cipher shifts each character forward by 11, which immediately tells us that `a` maps to `L`, and so on. A more general *simple substitution cipher* decouples the encryptions of different letters, e.g. each `a` maps to `C` and each `b` maps to `J`, etc.

Question 1.1. *Explain why this particular substitution cipher is not a shift cipher.*

Question 1.2. *How many possible keys are there in a substitution cipher? Hint: think of encryption as a function. What properties should this function have?*

What would cryptanalysis of a simple substitution cipher look like? There are more than 10^{26} keys in this case. If we could try a million keys every second, it would still take more than 10^{13} years to try them all, so the brute-force solution is infeasible. Despite the huge number of possible keys, simple substitution ciphers are often really easy to cryptanalyze in practice with simple *frequency analysis*. The idea is that if the plaintext is more than a few sentences long, then one might expect to see a lot of `e`'s, `t`'s and `a`'s and not many `z`'s or `q`'s. Consequently, if we look at the frequencies of the letters in the ciphertext, it would be reasonable to guess that the most common ciphertext letters correspond to the most common plaintext letters.

For example, suppose we intercept the following message.

```
LWNSOZ BNWVWB AYBNVB SQWVUO HWDIZW RBBNPB POOUWR PAWXAW
PBWZWM YPOBNP BBNWJP AWWRZS LWZQJB NVIAXA WPBSAL IBNXWA
BPIRYR POIWRP QOWAIE NBVBNP BPUSRE BNWVWP AWOIHW OIQWAB
JPRZBN WFYAVY IBSHNP FFIRWV VBNPBB SVWXYA WBNWVW AIENBV
ESDWAR UWRBVP AWIRVB IBYBWZ PUSREU WRZWAI DIREBH WIATYV
BFSLWA VHASUB NWXSRV WRBSHB NWESDW ARWZBN PBLNWR WDWAPR
JHSAUS HESDWA RUWRBQ WXSUWV ZWVBAY XBIDWS HBNWVW WRZVIB
IVBNVA IENBSH BNWFWS FOWBSP OBWASA BSPQSO IVNIBP RZBSIR
VBIBYB WRWLES DWARUW RBOPJI REIBVH SYRZPB ISRSRV YXNFAI
RXIFOW VPRZSA EPRIKI REIBVF SLWAVI RVYXNH SAUPVB SVWWUU
SVBOIC WOJBSW HHWXBB NWIAVP HWBJPR ZNPFFI RWVV
```

Let's arrange the letters in the ciphertext by frequency.

W	B	R	S	I	V	A	P	N	O	···
76	64	39	36	36	35	34	32	30	16	···

The letters in standard English text have the following frequencies.

E	T	A	O	N	R	I	S	H	D	···
.131	.105	.082	.080	.071	.068	.064	.061	.053	.038	···

Since the letter W appears much more frequently than the other letters in the ciphertext, it tips us off that we might be dealing with a substitution cipher and that an e in the plaintext probably maps to a W in the ciphertext. It's also reasonable to guess that the letters B, R, S and I correspond to the letters t, a, o and i in some order.

Looking at individual letter frequencies lets us get our foot in the door, but it doesn't help us much when it comes to differentiating between letters that appear with roughly the same frequency (like R and S in this ciphertext). If we think about English text for a bit, we notice that certain pairs of letters, called *bigrams*, appear together more frequently than others (e.g. q is almost always followed by a u and th is a common pair). Here are a few of the bigram frequencies from our ciphertext

	W	B	R	S	I	V	A	P	N
W	3	4	12	2	4	10	14	3	1
B	4	4	0	11	5	5	2	4	20
R	5	5	0	1	1	5	0	3	0
S	1	0	5	0	1	3	5	2	0
I	1	8	10	1	0	2	3	0	0
V	8	10	0	0	2	2	0	3	1
A	7	3	4	2	5	4	0	1	0
P	0	8	6	0	1	1	4	0	0
N	14	3	0	1	1	1	0	7	0

That is, this table tells us that WN appears once and NW appears 14 times. In English, the letter h frequently comes before e and rarely comes after it, so it's a safe guess that h maps to N in this particular substitution. Since th is the most common digram in English and BN is the most common digram in the ciphertext, we guess that t maps to B. Other features of the English language lead to more educated guesses that lead to a full cryptanalysis of the ciphertext.

Problem 1.3. Finish decrypting the ciphertext. One place to start is by looking for vowels and noting that some vowels like a, i and o tend to avoid each other.

1.2 Divisibility and Greatest Common Divisors

Some of the most widely-used cryptosystems today make heavy use of abstract algebra and number theory. Roughly speaking, number theory is concerned with properties of the integers, $\mathbb{Z}$, like divisibility and solutions to equations with integer variables.

Definition 1.4. Let a and b be integers with $b \neq 0$. We say that b *divides* a if $a = bc$ for some integer c, in which case, we write $b \mid a$.

Example 1.5. (a) We call the integers divisible by 2 *even* and those that aren't *odd*. Is zero even or odd?

(b) 713 is divisible by 23 since $713 = 23 \cdot 31$. The numbers used in everyday cryptographic applications are hundreds or even thousands of digits long.

(c) A number n is divisible by 5 if and only if it ends in a 0 or a 5 (when written in base 10, of course). To see this, write

$$n = d_0 + 10d_1 + 10^2 d_2 + \cdots + 10^k d_k,$$

where $k \geq 0$ and $d_i \in \{0, 1, 2, \ldots, 9\}$ for all i. Then d_0 is the number that n "ends" with, so if it's 0 or 5, we can just factor a 5 out of the right-hand side to see that n is divisible by 5. Conversely, if we rearrange this,

$$d_0 = n - 10d_1 - 10^2 d_2 - \cdots - 10^k d_k,$$

we see that if n is divisible by 5, then the whole right-hand side (which is equal to d_0) is also divisible by 5.

We record some basic properties of divisibility here. The proof of this proposition is a straightforward exercise.

Proposition 1.6. *Let a, b and c be integers.*

(a) If $a \mid b$ and $b \mid c$, then $a \mid c$.

(b) If $a \mid b$ and $b \mid c$, then $a = \pm b$.

(c) If $a \mid b$ and $a \mid c$, then $a \mid (b + c)$ and $a \mid (b - c)$.

Question 1.7. *For those familiar with equivalence relations, is divisibility an equivalence relation on $\mathbb{Z}$?*

Definition 1.8. A *common divisor* of integers a and b is a positive integer d that divides both of them. The *greatest common divisor* of a and b is the largest positive integer d such that $d \mid a$ and $d \mid b$ and we write $d = \gcd(a, b)$ or $d = (a, b)$ if there is no possibility of confusion.

Example 1.9. (a) Find the greatest common divisor of 132 and 66 by listing out all of their divisors.

(b) Find the greatest common divisor of 80 and 5. Other than the number being pretty small, why was this easy to do? Prove your idea.

Of course given integers a and b, it's not always the case that $a \mid b$ or $b \mid a$. In this case, we get a (unique) remainder.

Proposition 1.10. *For any positive integers a and b, there exist unique integers q and r such that*

$$a = bq + r \qquad \text{with } 0 \leq r < b. \tag{1}$$

Here we call q the quotient *and r the* remainder *when a is divided by b.*

Proof. Homework exercise. □

Division with remainder provides us with a way of finding the gcd of two integers. To see this, rearrange (1) to obtain

$$r = a - bq.$$

If d is a common divisor of a and b, then it clearly divides the right-hand side of this equation, so it must divide r as well. A similar rearrangement (which?) shows that if c is a common divisor of b and r, then it must also divide a. We then have that the common divisors of a and b are the common divisors of b and r, so we must have that

$$\gcd(a, b) = \gcd(b, r).$$

This is great because if we assume that $a > b$, then we've reduced the problem of finding $\gcd(a, b)$ to finding the gcd of two smaller numbers, b and r. We can then repeat this: divide b by r to obtain

$$b = q'r + r', \qquad \text{with } 0 \leq r' < r.$$

By the same reasoning, we have that

$$\gcd(a, b) = \gcd(b, r) = \gcd(r, r').$$

Since the remainders are positive numbers that get strictly smaller after each division, we must eventually reach a remainder of zero. The remainder right before this one is then the gcd of a and b.

Example 1.11. Let's compute $\gcd(12345, 11111)$. Even without a calculator it's sometimes easy to eyeball how many times one number goes into another.

$$\begin{aligned} 12345 &= 11111 \cdot 1 + 1234 \\ 11111 &= 1234 \cdot 9 + 5 \\ 1234 &= 5 \cdot 246 + 4 \\ 5 &= 4 \cdot 1 + 1 \\ 4 &= 1 \cdot 4 + 0 \end{aligned}$$

The second-to-last remainder we found was 1, so we conclude that $\gcd(12345, 11111) = 1$. Note that even though the numbers involved started out somewhat large (for by-hand computations), we were able to calculate the gcd in just a few steps.

This procedure for computing the gcd of two integers is called the *Euclidean algorithm* after the ancient Greek mathematician. We summarize it here.

Theorem 1.12. *Let $a \geq b$ be positive integers. Then the following algorithm computes $\gcd(a, b)$ in a finite number of steps (i.e., the algorithm eventually terminates).*

1: Let $r_0 = a$ and $r_1 = b$.
2: Set $i = 1$.
3: Divide r_{i-1} by r_i with remainder to obtain quotient q_i and remainder r_{i+1}.

$$r_{i-1} = r_i \cdot q_i + r_{i+1}, \qquad \text{with } 0 \leq r_{i+1} < r_i.$$

4: If $r_{i+1} = 0$, then $r_i = \gcd(a, b)$ and the algorithm terminates.
5: Otherwise, $r_{i+1} > 0$. Set $i = i + 1$ and go to Step 3.

How many times do we need to repeat the division step of the algorithm? Let's start by looking at how much the remainders drop at each step. At each step we have two possibilities: either $r_{i+1} \leq \frac{1}{2} r_i$ or $r_{i+1} > \frac{1}{2} r_i$. In the first case, since the remainders are strictly decreasing, we have

$$r_{i+2} < r_{i+1} \leq \frac{1}{2} r_i.$$

In the other case we must have $r_i = r_{i+1} \cdot 1 + r_{i+2}$. Rearranging, we have

$$r_{i+2} = r_i - r_{i+1} < r_i - \frac{1}{2} r_i = \frac{1}{2} r_i.$$

In either case, we have that the remainder drops by at least half every two steps. After $2k+1$ steps we then have

$$r_{2k+1} < \frac{1}{2} r_{2k-1} < \frac{1}{2^2} r_{2k-3} < \cdots < \frac{1}{2^k} r_1 = \frac{1}{2^k} b.$$

If k is the smallest integer such that $b/2^k < 1$, then we have $r_{2k+1} = 0$. Setting $k = \lfloor \log_2 b \rfloor + 1$ does the trick. The *gcd* is then found on step at most $2k = 2\lfloor \log_2 b \rfloor + 2$.

Remark 1.13. Pretty much all cryptography software includes some implementation of the Euclidean algorithm. Computers store integers in their binary representations where an integer N takes $n = \lfloor \log_2 N \rfloor + 1$ bits of memory (why?). The above analysis shows that the Euclidean algorithm runs in a number of steps equal to at most twice the number of bits ($2n$) it takes to store the smaller of its two inputs. When the number of steps it takes an algorithm to complete grows (at most) like a polynomial in its input size, then we consider it to be (reasonably) efficient.

The Euclidean algorithm also gives us a way of writing $\gcd(a, b)$ as a linear combination of a and b.

Example 1.14. Let's return to Example 1.11.

Write $a = 12345$ and $b = 11111$ and solve for the first remainder, 1234, in terms of a and b:

$$1234 = a - b.$$

Now plug this into the second equation to get

$$b = (a - b) \cdot 9 + 5,$$

So the next remainder, 5, can be written in terms of a and b as

$$5 = -9a + 10b.$$

Plug this along with the expression for 1234 into the third equation to get

$$a - b = (-9a + 10b) \cdot 246 + 4,$$

which gives the next remainder, 4, in terms of a and b:

$$4 = 2215a - 2461b.$$

Finally, plug the expressions for 4 and 5 into the second-to-last equation to get

$$1 = (-9a + 10b) - (2215a - 2461b) = -2224a + 2471b.$$

This example is more or less a proof of the following theorem.

Theorem 1.15. *Let a and b be positive integers. Then the equation*

$$ax + by = c$$

has integer solutions for x and y if and only if c is divisible by $\gcd(a, b)$. Moreover, if (x_0, y_0) is a particular solution to this equation, then every other solution has the form

$$x = x_0 + \frac{kb}{\gcd(a, b)}, \quad y = y_0 - \frac{ka}{\gcd(a, b)}$$

for some integer k.

1.3 Modular Arithmetic

Recall that when encrypting a message with a shift cipher with key k, each letter in the plaintext is shifted forward in the alphabet by k positions. Importantly, we *wrap around* the alphabet if we shift past the letter `Z` (or whatever letter is at the end of the relevant alphabet). This idea of wrapping around the end back to the beginning comes up in our day-to-day lives when we think about telling time. Four hours after 9AM is 1PM since we *wrap around* 12pm back to 1PM (the same idea holds if you prefer to think in military time - three hours after 2300 is 0200). We'll look at this mathematically with the idea of *congruence*.

Definition 1.16. Let $m \geq 1$ be an integer. We say that the integers a and b are *congruent modulo* m if the difference $a - b$ is divisible by m. In this case we write

$$a \equiv b \pmod{m}$$

and call m the *modulus*.

Example 1.17. We have that $2 \equiv 5 \pmod{7}$. We also have that $2 \equiv 9 \pmod{7}$ and $2 \equiv 16 \pmod{7}$.

Importantly, congruences respect familiar operations like addition and multiplication, but are a little trickier when it comes to division.

Proposition 1.18. *Let $m \geq 1$ be an integer.*

1. *If $a_1 \equiv a_2 \pmod{m}$ and $b_1 \equiv b_2 \pmod{m}$, then*

$$a_1 \pm b_1 \equiv a_2 \pm b_2 \pmod{m} \qquad \text{and} \qquad a_1 b_1 \equiv a_2 b_2 \pmod{m}.$$

2. *Let a be an integer. Then there exists an integer b such that*

$$ab \equiv 1 \pmod{m}$$

if and only if $\gcd(a, m) = 1$. In this case, we call b the multiplicative inverse of a modulo m *and we write $b = a^{-1} \pmod{m}$.*

Proof. The proof of part (a) isn't super interesting, so we'll skip it.

For part (b), first suppose that $\gcd(a, m) = 1$. Then by Theorem (1.15), we can find u and v such that $au + mv = 1$. But if we rearrange this, we have

$$au - 1 = mv,$$

so the difference $au - 1$ is divisible by m and $au \equiv 1 \pmod{m}$. In this case, u is an inverse of $a \pmod{m}$.

On the other hand, suppose a has a multiplicative inverse $b \pmod{m}$. Then m divides the difference $ab - 1$ so we have

$$ab - km = 1$$

for some integer k. If d is some (positive) common divisor of a and m, then d must divide the left-hand side of this equation. But then d must divide 1, so we must have $d = 1$. It must then be the case that $\gcd(a, m) = 1$. □

Part (b) of this proposition gives us a partial analogue of division modulo m. Just like how the rational number $1/2$ has the property that $(1/2) \cdot 2 = 1$, the number 3 has the property that $3 \cdot 2 = 6 \equiv 1 \pmod{5}$, so 3 plays a similar role to $1/2$. What's more is that our proof of part (b) gives us an algorithm for computing the modular inverse: the extended Euclidean algorithm.

Example 1.19. Let's find the inverse of 4 modulo 7 (if it exists at all). First compute $\gcd(4, 7)$.

$$\begin{aligned} 7 &= 4 \cdot 1 + 3 \\ 4 &= 3 \cdot 1 + 1 \\ 3 &= 1 \cdot 3 + 0 \end{aligned}$$

So $\gcd(4, 7) = 1$, so we know a modular inverse exists. We find it by substituting in expressions for the remainders.

$$\begin{aligned} 1 &= 4 - 3 \cdot 1 \\ &= 4 - (7 - 4) \\ &= 4 \cdot 2 - 7. \end{aligned}$$

Rearranging this, we see that $4 \cdot 2 - 1 = 7$, so $4 \cdot 2 \equiv 1 \pmod{7}$, and 2 is the inverse of 4 modulo 7.

Remember that the Euclidean algorithm is really efficient (for a computer at least - so is the extended one), so finding inverses is efficient as well.

Returning to the above example, note that $4 \cdot 9 = 36 \equiv 1 \pmod{7}$ as well, so we can just as easily say that 9 is an inverse of 4 modulo 7. It would be nice if there was just one inverse or a way for two people to pick the same inverse every time. Division with remainder gives us a way of doing this. If b is an inverse of a modulo m, write

$$b = mq + r \quad \text{with } 0 \leq r < m.$$

Then r is always between 0 and $m-1$. Since this r is unique, we can agree that we always work with the integers 0 through $m - 1$ when we work modulo m. This idea is encapsulated in the following proposition.

Proposition 1.20. *The integers a and b are congruent modulo m if and only if they have the same remainder when divided by m.*

Recall the notion of equivalence relations.

Definition 1.21. A relation $\sim$ on a set X is an *equivalence relation* if the following all hold.

1. (Reflexivity) $x \sim x$ for all $x \in X$.
2. (Symmetry) $x \sim y$ if and only if $y \sim x$ for any $x, y \in X$.
3. (Transitivity) if $x \sim y$ and $y \sim z$ then $x \sim z$.

For each $x \in X$, the *equivalence class of* x, denoted $[x]$ (or sometimes $\overline{x}$) is

$$[x] = \{y \in X : x \sim y\}.$$

We can form the new set $X/\sim$, the *quotient of* X *by* $\sim$ by just taking equivalence classes.

$$X/\sim = \{[x] : x \in X\}.$$

Modular arithmetic is a concrete example of this.

Proposition 1.22. *Fix a positive integer $m \geq 2$. Then equivalence modulo m is an equivalence relation on $\mathbb{Z}$.*

Moreover, Proposition (1.20) leads us to think that the quotient of $\mathbb{Z}$ by "equivalence modulo m" is the "correct" object to work with and to choose our equivalence classes to be $[0], \ldots, [m-1]$.

Definition 1.23. The set $\mathbb{Z}/m\mathbb{Z}$ is defined to be the set of integers quotiented by the relation "equivalent modulo m". Specifically,

$$\mathbb{Z}/m\mathbb{Z} = \{[0], \ldots, [m-1]\},$$

where $[a] = \{b \in \mathbb{Z} : a \equiv b \pmod{m}\}$.

Remark 1.24. When working with $\mathbb{Z}/m\mathbb{Z}$, we usually drop the $[\cdot]$ when talking about its elements, which are equivalence classes. That is, it technically doesn't make sense to write $2 \in \mathbb{Z}/5\mathbb{Z}$ since 2 isn't an equivalence class. However, as the next proposition shows, the equivalence class $[2]$ can be made to behave a lot like the ordinary integer 2.

We can carry the notions of addition and multiplication over to the quotient as well.

Definition 1.25. For $[a], [b] \in \mathbb{Z}/m\mathbb{Z}$, define $[a] + [b]$ to be $[a+b]$ and $[a] \cdot [b]$ to be $[ab]$.

Remark 1.26. Technically, the above definition should be made into a proposition that says this definition is *well-defined*. That is, we need to show that if $a \equiv a'$ and $b \equiv b'$ then we want $[a] + [b] = [a'] + [b']$ and $[a] \cdot [b] = [a'] \cdot [b']$. This follows easily from Proposition (1.18).

Let's think about Theorem 1.15 for a bit in this context by looking at equations in $\mathbb{Z}/m\mathbb{Z}$.

Example 1.27. 1. Does $2x \equiv 3$ have a solution modulo 5? It would be nice if we could "divide by 2" and that's exactly what a multiplicative inverse lets us do. It's easy to verify that 4 is the inverse of 2 (mod 5), so multiplying both sides of this equation through by 4 gives $x \equiv 12 \equiv 2 \pmod{5}$.

2. What about $2x \equiv 3 \pmod 6$? This equation in $\mathbb{Z}/6\mathbb{Z}$ is equivalent to the integer equation $2x - 3 = 6y$, which has no solution since the left-hand side is always odd while the right-hand side is always even. Another way we can think about it is that we can't "divide by 2" since $\gcd(2, 6) = 2 \neq 1$.

3. What about $2x \equiv 4 \pmod 6$? Just like in the last example, we can't divide by 2. However, it's easy to see that $x \equiv 2 \pmod 6$ is a solution. But this solution isn't unique since $x \equiv 5 \pmod 6$ is also a solution.

In short, the existence of an inverse, as determined by Theorem 1.15 determines whether or not equations like $ax \equiv b \pmod m$ have solutions. If $\gcd(a, m) = 1$, then there's a unique solution. Otherwise, there can either be no solution or there might be multiple solutions. If we want to restrict ourselves to the (equivalence classes of) integers that *do* have inverses modulo m, then we use the following object.

Definition 1.28. Fix an integer $m \geq 2$. Then the set of *units modulo* m is denoted by

$$\begin{aligned}(\mathbb{Z}/m\mathbb{Z})^\times &= \{a \in \mathbb{Z}/m\mathbb{Z} : \gcd(a, m) = 1\} \\ &= \{a \in \mathbb{Z}/m\mathbb{Z} : a \text{ has an inverse modulo } m\}.\end{aligned}$$

1.4 Prime Numbers, Unique Factorization, and Finite Fields

The "building blocks" of the integers are the prime numbers.

Definition 1.29. An integer p is called *prime* if $p \geq 2$ and if the only positive integers dividing p are 1 and p.

Note that if p is prime, then $\gcd(a, p) = 1$ for each $1 \leq a < p$ (why?). Consequently, each nonzero element of $\mathbb{Z}/p\mathbb{Z}$ has a multiplicative inverse, i.e.

$$(\mathbb{Z}/p\mathbb{Z})^\times = \{1, 2, \ldots, p - 1\}.$$

The set $\mathbb{Z}/p\mathbb{Z}$ forms a structure that we call a (finite) *field*: a set where we can add and subtract as well as multiply and divide by (nonzero) elements. We denote this field by $\mathbb{F}_p$. Other examples of fields include $\mathbb{R}$ and $\mathbb{Q}$ but not $\mathbb{Z}$.

Proposition 1.30. *Let p be a prime number and suppose that $p \mid ab$. Then $p \mid a$ or $p \mid b$. More generally, if*

$$p \mid a_1 a_2 \cdots a_k,$$

then $p \mid a_i$ for some i.

Proof. We'll prove the first statement and you'll prove the second one in discussion. If p divides a then we're done. If $p \nmid a$, then $\gcd(a, p) = 1$ (why?), so we can write

$$au + pv = 1$$

for some integers u and v. Multiplying this through by b gives

$$abu + pbv = b.$$

By assumption, $p \mid ab$ and clearly $p \mid pbv$, so p divides the left-hand side of this equation. Consequently, p divides the right-hand side, which is b. □

Using this, we can prove what we said earlier about primes being "building blocks."

Theorem 1.31 (The Fundamental Theorem of Arithmetic)**.** *Let $a \geq 2$ be an integer. Then a can be factored as a product of prime numbers*

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} \tag{2}$$

for some positive integer r. Furthermore, this factorization is unique up to rearrangement of the primes.

Proof. We prove that we can factor into primes by induction and uniqueness will come later. Our base case is $a = 2$, and this itself is a prime factorization since 2 is prime. Suppose then that every integer less than a can be factored into primes. If a itself is prime, then we're done by the same reasoning we used in the base case. Otherwise, $a = bc$ where $1 < b, c < a$. By the induction hypothesis, we can factor b and c into primes:

$$b = p_1^{e_1} \cdots p_k^{e_k}, \quad c = q_1^{f_1} \cdots q_\ell^{f_\ell}.$$

But then $a = p_1^{e_1} \cdots p_k^{e_k} q_1^{f_1} \cdots q_\ell^{f_\ell}$ is a factorization of a. You'll prove the uniqueness part of this statement in discussion. □

Looking at the factorization of a into primes (2), we call the number of times a particular prime, p, appears in the factorization the *order of p in a* and denote it by $ord_p(a)$. That is, in the factorization (2), $ord_{p_i}(a) = e_i$.

1.5 Powers and Primitive Roots in Finite Fields

We can add, subtract, multiply and (sometimes) divide by elements of $\mathbb{Z}/m\mathbb{Z}$. Since we can multiply, we can definitely exponentiate in exactly the way you think we would. If $a \in \mathbb{Z}/m\mathbb{Z}$ and k is a nonnegative integer, then a^k is the product of a with itself k times, taken modulo k. Moreover, if a has inverse a^{-1}, then we can define negative powers of a as positive powers of a^{-1}. We need to be a little careful though. We can raise an element of $\mathbb{Z}/m\mathbb{Z}$ to an integer power, but it doesn't make sense to raise an element of $\mathbb{Z}/m\mathbb{Z}$ to the power of another element of $\mathbb{Z}/m\mathbb{Z}$.

Example 1.32. We clearly have that $2^1 \equiv 2 \pmod 5$. However, $2^5 = 32 \equiv 2 \pmod 5$ even though $5 \not\equiv 1 \pmod 5$. In other words, $a^b \equiv a^c \pmod m$ *does not* imply that $b \equiv c \pmod m$.

A few of the main cryptographic protocols we'll talk about come from the properties of modular exponentiation, so let's talk a bit about it. Let's look at some of the powers of 2 modulo 7

$$2^1 \equiv 2,\ 2^2 \equiv 4,\ 2^3 = 8 \equiv 1,\ 2^4 = 16 \equiv 2,\ 2^5 = 32 \equiv 4,\ 2^6 = 64 \equiv 1, \ldots$$

It looks like we get a repeating pattern of 1, 2, 4. In fact, we can prove that this pattern holds true: take any positive integer k and divide it by 3 with remainder to get $k = 3q + r$. Then

$$2^k = 2^{3q+r} = (2^3)^q \cdot 2^r \equiv 1^q \cdot 2^r \equiv 2^r \pmod 7.$$

That is, the value of 2^k only depends on the remainder we get when we divide k by 3, i.e. we care about what k is modulo 3, *not* modulo 7. What happens with the powers of other numbers, say 3 (still modulo 7)?

$$3^1 = 3,\ 3^2 \equiv 2,\ 3^3 \equiv 6,\ 3^4 \equiv 4,\ 3^5 \equiv 6,\ 3^6 \equiv 1,\ 3^7 \equiv 3, \ldots$$

Like with 2, we have that $3^6 \equiv 1 \pmod 7$. However, it looks like we get a repeating pattern of length six this time. What's more is that the powers of 3 give us all the nonzero elements of $\mathbb{Z}/7\mathbb{Z}$.

Let's solidify the first of these observations into a theorem.

Theorem 1.33 (Fermat's Little Theorem)**.** *Let p be a prime number and let a be any integer. Then*

$$a^{p-1} \equiv \begin{cases} 1 \pmod{p}, & \text{if } p \nmid a, \\ 0 \pmod{p}, & \text{if } p \mid a. \end{cases}$$

Proof. If p divides a then it divides every power of a, so let's just look at the case where $p \nmid a$. Let's look at the numbers

$$a,\ 2a,\ 3a,\ \ldots,\ (p-1)a, \tag{3}$$

We claim that these are all *distinct* when reduced modulo p. Indeed, if $ka \equiv ja \pmod{p}$, then p divides $a(k-j)$. By Proposition 1.30, p must then divide a or $k-j$. Since we've assumed that $p \nmid a$, we must have that p divides $k-j$. But we haven't listed any multiples of p above, so we must have $k = j$.

Now let's multiply all the elements in (3) together. On one hand, this is clearly $a^{p-1} \cdot (p-1)!$. On the other hand, since since these are $p-1$ distinct nonzero integers between 1 and $p-1$, the must be all of the integers in this range, so their product is $(p-1)!$. We must then have

$$a^{p-1} \cdot (p-1)! \equiv (p-1)! \pmod{p}.$$

We can then cancel the $(p-1)!$ from both sides (why?) to obtain $a^{p-1} \equiv 1 \pmod{p}$. □

This theorem has some really powerful implications for computation.

Example 1.34. The integer $p = 15485863$ is prime, so by Fermat's little theorem we have

$$2^{15485862} \equiv 1 \pmod{15485863}.$$

Even though the numbers involved are large ($2^{15485862}$ has more than 400,000 digits), we can write the above identity without doing any real computation (we had to know that 15485863 is prime first, and we'll see some good algorithms for verifying this later).

Fermat's little theorem tells us that if $p \nmid a$, then $a^{p-1} \equiv 1 \pmod{p}$, but as we saw with the powers of 2 modulo 7, a smaller power of a might be congruent to 1 modulo p. This motivates the following definition.

Definition 1.35. Let $p \geq 2$ be prime. For any integer a such that $p \nmid a$, the *order of a modulo p* is the smallest positive integer k such that $a^k \equiv 1 \pmod{p}$.

Fermat's little theorem tells us that that the order of a is at most $p-1$ so long as $p \nmid a$. The following proposition claims that the order of a modulo p can't be just anything however.

Proposition 1.36. *Let p be a prime and let a be an integer with $p \nmid a$. If $a^n \equiv 1 \pmod{p}$, then the order of a modulo p divides n. In particular, the order of a divides $p-1$.*

Proof. Suppose k is the order of a modulo p. We divide n by k to obtain

$$n = kq + r$$

for some $0 \leq r < k$. We then have

$$1 \equiv a^n \equiv a^{kq+r} \equiv (a^k)^q \cdot a^r \equiv 1^q \cdot a^r \equiv a^r \pmod{p}.$$

But k is the smallest positive power of a congruent to 1 modulo p, so we must have $r = 0$ and $k \mid n$. □

Looking at the powers of 3 modulo 7, we see that sometimes the powers of a modulo p can fill out all of the nonzero residues modulo p. The following theorem says that there is always such a p and you'll prove it on your next homework assignment.

Theorem 1.37 (Primitive Root Theorem)**.** *Let p be prime. Then there exists an element $g \in \mathbb{F}_p^\times$ such that*

$$\mathbb{F}_p^\times = \{1, g, g^2, \ldots, g^{p-2}\}.$$

Such a g is called a generator *or* primitive root of $\mathbb{F}_p$.

Let's talk a little about how to actually compute $a^k \pmod m$. The naive way, repeated multiplication, would simply compute a^i for all $i \leq k$:

$$a_1 \equiv a \pmod m, \quad a_2 \equiv a \cdot a_1 \pmod m, \quad a_3 \equiv a \cdot a_2 \pmod m, \quad \ldots \quad a_k = a \cdot a_{k-1} \pmod m.$$

If k is of moderate size (for a computer), say around 1000 bits (around 300 digits), then the time it would take to complete this algorithm would be greater than the estimated age of the universe, even if you reduced modulo m after each step (if you didn't, then the integer a^k would take up more bits than there are particles in the universe by some estimates). Let's look at an example for how to compute large powers very efficiently.

Example 1.38. Let's compute $3^{75} \pmod{100}$. We start by writing 75 in binary. The largest power of 2 that is no larger than 75 is 64, so

$$75 = 64 + 11.$$

Now the largest power of 2 at most 11 is 8. We repeat this process.

$$\begin{aligned} 76 &= 64 + 8 + 3 \\ &= 64 + 8 + 2 + 1. \end{aligned}$$

Using this we can write

$$3^{76} = 3^{64+8+2+1} = 3^{64} \cdot 3^8 \cdot 3^2 \cdot 3^1. \tag{4}$$

Now we can compute the seven numbers

$$3,\ 3^2,\ 3^4,\ 3^8,\ \ldots,\ 3^{64}$$

modulo 100 quite easily - each number is just the square of the one before it. Now using (4), we decide which of these powers of 3 to multiply together.

i	0	1	2	3	4	5	6
$3^{2^i} \pmod{100}$	3	9	81	61	21	41	81

This gives

$$\begin{aligned} 3^{76} &= 3^1 \cdot 3^2 \cdot 3^8 \cdot 3^{64} \\ &\equiv 3 \cdot 9 \cdot 61 \cdot 81 \pmod{100} \\ &\equiv 21 \pmod{100}. \end{aligned}$$

Let's describe this algorithm, sometimes called the *fast powering algorithm* or the *square-and-multiply algorithm*, more formally.

1. **Given:** integers g, A and N
2. Compute the binary expansion of A as
$$A = A_0 + A_1 \cdot 2 + A_2 \cdot 2^2 + \cdots + A_r \cdot 2^r, \quad \text{with } A_i \in \{0, 1\} \text{ for all } i.$$
3. Compute the powers $A^{2^i} \pmod m$ for each $0 \leq i \leq r$ by squaring.
$$\begin{aligned} a_0 &\equiv g \pmod N \\ a_1 &\equiv a_0^2 \equiv g^2 \pmod N \\ a_2 &\equiv a_1^2 \equiv g^{2^2} \pmod N \\ &\vdots \\ a_r &\equiv a_{r-1}^2 \equiv g^{2^r} \pmod N. \end{aligned}$$
4. Compute $g^A \mod N$ by multiplication.
$$\begin{aligned} g^A &= g^{A_0 + A_1 \cdot 2 + A_2 \cdot 2^2 + \cdots + A_r \cdot 2^r} \\ &= g^{A_0} \cdot (g^2)^{A_1} \cdot (g^{2^2})^{A_2} \cdots (g^{2^r})^{A_r} \\ &\equiv a_0^{A_0} \cdot a_1^{A_1} \cdots a_r^{A_r} \pmod n. \end{aligned}$$

1.7 Symmetric and Asymmetric Ciphers

Let's briefly formalize some notions from cryptography. Suppose Alice and Bob agree to send and receive messages using some cipher. If the same key is used for encryption and decryption, then we say that the cipher is *symmetric*. We can describe a symmetric cipher with these sets and maps.

$$\begin{aligned} \mathcal{K} &= \text{all possible keys} \\ \mathcal{M} &= \text{all possible plaintexts} \\ \mathcal{C} &= \text{all possible ciphertexts} \\ e &: \mathcal{K} \times \mathcal{M} \to \mathcal{C} \\ d &: \mathcal{K} \times \mathcal{C} \to \mathcal{M}. \end{aligned}$$

The function e is the *encryption function* that takes in a key, plaintext pair (k, m) and outputs the corresponding ciphertext. Likewise, the *decryption function* d takes a key, ciphertext pair (k, c) and outputs the plaintext.

Of course we want decryption to undo encryption with the same key, so we require the following consistency condition.

$$d(k, e(k, m)) = m, \qquad \text{for all } k \in \mathcal{K},\ m \in \mathcal{M}.$$

Another way to think about this condition is to define encryption and decryption functions e_k, d_k for each key k,

$$e_k(\cdot) = e(k, \cdot), \qquad d_k(\cdot) = d(k, \cdot)$$

Then the consistency condition just says that d_k is the inverse function of e_k. This of course requires that e_k be injective.

Example 1.39. The substitution cipher is a symmetric cipher.

When assessing the security of a cipher, we *always assume that an eavesdropper, Eve, knows the method of encryption.* That is, we assume Eve knows e and d, but not the key k. Under this assumption, here are some of the properties we want for $(\mathcal{K}, \mathcal{M}, \mathcal{C}, e, d)$.

1. Easy encryption: for any $k \in \mathcal{K}$ and $m \in \mathcal{M}$, it's easy to compute $e_k(m)$.
2. Easy decryption: for any $k \in \mathcal{K}$ and $c \in \mathcal{C}$, it's easy to compute $d_k(c)$.
3. Strength against ciphertext-only attacks: If Eve knows that the ciphertexts $c_1, c_2, \ldots, c_n \in \mathcal{C}$ were encrypted using the same (unknown) key k, it should be hard for her to compute any of the corresponding plaintexts, $d_k(c_1), \ldots, d_k(c_n)$ without knowing k.
4. Strength against known-plaintext attacks: If Eve knows some plaintexts and their corresponding ciphertexts $(m_1, c_1), \ldots, (m_n, c_n)$, where $c_i = e_k(m_i)$, then it should be hard for her to compute $d_k(c)$ for any new ciphertext c without knowing k.
5. Strength against chosen-plaintext attacks: If Eve *chooses* some plaintexts $m_1, \ldots, m_n$ and knows their corresponding ciphertexts $c_1, \ldots, c_n$, it should still be hard for her to compute $d_k(c)$ for any new ciphertext c without knowing k.
6. Strength against chosen-ciphertext attacks: If Eve chooses some ciphertexts $c_1, \ldots, c_n$ and knows their corresponding plaintexts $m_1, \ldots, m_n$, it should still be hard for her to compute $d_k(c)$ for any new ciphertext c without knowing k.

Example 1.40. The substitution cipher isn't very resistant to ciphertext-only attacks since it's vulnerable to frequency analysis when we have a long ciphertext. It's also quite vulnerable to known-plaintext attacks since we can just query a few messages to learn most of the substitutions.

Remark 1.41. When we think of plaintexts we often think of strings of Latin characters. However, we can usually *encode* written characters into numbers (indeed, this is what computers do), so we can safely think of $\mathcal{M}$ and $\mathcal{C}$ as sets of numbers or elements of $\mathbb{Z}/m\mathbb{Z}$.

Example 1.42. Let $\mathcal{M} = \mathcal{C} = \mathcal{K} = \mathbb{F}_p^\times$. For any key k, define encryption by

$$e_k(m) = m \cdot k \pmod{p}.$$

We can assume that basic arithmetic like addition and multiplication modulo p is efficient to compute. We can also efficiently decrypt, since the Euclidean algorithm gives us a way to compute $k^{-1} \pmod{p}$ in around $2 \log_2 p$ steps, so

$$d_k(c) = k^{-1} \cdot c \pmod{p}$$

is efficient to compute. This cipher is *completely broken* (that is, we can learn the key) by a known plaintext attack. If we know c and m, then we can simply compute

$$k \equiv c^{-1} \cdot m \pmod{p}$$

since we can efficiently compute $c^{-1} \pmod{p}$.

One issue with symmetric ciphers is how Alice and Bob agree on a key. If they have to exchange the key in the open, then it's vulnerable to Eve. Alternatively, they can meet in person, but this can be inconvenient or impossible. This motivates the idea of *public-key cryptography*. Here, we have two separate keys, $k = (k_{priv}, k_{pub})$. The *public key*, k_{pub} is known to everyone, while the *private key* is known only to the person receiving messages. We have encryption and decryption functions

$$e_{k_{pub}} : \mathcal{M} \to \mathcal{C}$$
$$d_{k_{priv}} : \mathcal{C} \to \mathcal{M},$$

that are inverse to one another. It should be difficult to compute $d_{k_{priv}}$ without knowledge of k_{priv} even if one knows k_{pub}.

If Bob wants to send Alice a message, then Alice first sends Bob her public key, k_{pub} (or maybe she publishes it in some public place like some website). Note that we should then assume that Eve knows k_{pub} as well. Bob then computes $e_{k_{pub}}(m)$ and sends it to Alice. Alice then decrypts this using k_{priv}, which only she knows. The private key k_{priv} is sometimes called *trapdoor information* for the *one-way* function $e_{k_{pub}}$ because it should be hard to invert this function without knowledge of k_{priv}.

2 Discrete Logarithms and Diffie-Hellman

2.2 The Discrete Logarithm Problem

Public-key cryptography revolves around functions that are easy to compute but hard to invert without some special trapdoor information. We've seen that exponentiation modulo p is easy to compute by the square-and-multiply algorithm and invertible by the primitive root theorem. It turns out that it's also hard to invert.

Definition 2.1. Let g be a primitive root of $\mathbb{F}_p^\times$ and let $h \neq 0$ in $\mathbb{F}_p$. The *discrete logarithm problem (DLP)* is the problem of finding an exponent x such that

$$g^x \equiv h \pmod{p}.$$

The number x is called the *discrete logarithm of h to the base g* and is defined modulo $p-1$ by Fermat's little theorem.

Remember that since g is a primitive root we have that

$$\mathbb{F}_p^\times = \{g^0, g^1, \ldots, g^{p-2}\},$$

so $h \equiv g^i$ for some $0 \leq i \leq p-2$. By Fermat's little theorem we have that $g^{p-1} \equiv 1 \pmod{p}$, so

$$g^{i+k(p-1)} \equiv h \pmod{p}$$

for any integer k. This[2] is why we say that the discrete logarithm is defined modulo $p-1$. In particular, there are infinitely many integers x such that $g^x \equiv h \pmod{p}$ and the DLP asks us to find just one such x. This motivates us to define the function

$$\log_g : \mathbb{F}_p^\times \to \mathbb{Z}/(p-1)\mathbb{Z}$$

[2]If you've taken complex analysis, this might remind you of the fact that $e^{r+2\pi i} = e^r$ in $\mathbb{C}^\times$ and why we have to take so-called *branch cuts* of the logarithm in that setting.

for any primitive root g.

It makes sense to call this function a logarithm because it behaves how you would hope a logarithm does.

Proposition 2.2. *For any primitive root g of $\mathbb{F}_p^\times$ and any $a, b \in \mathbb{F}_p^\times$ we have that*

$$\log_g(ab) \equiv \log_g(a) + \log_g(b) \pmod{p-1}.$$

Proof. You'll prove this in your discussion section. □

How do we compute discrete logarithms? Let's look at the brute-force solution. For any $A \in \mathbb{Z}/(p-1)\mathbb{Z}$ we can compute $g^A \pmod{p}$ in around $2\log_2 p$ multiplications (steps), and if try to solve the DLP by trial and error, we'd compute

$$g^0,\ g^1,\ g^2, \ldots$$

modulo p until we find h. If $h \equiv g^{p-2}$ then this will take around $2p\log_2 p$ steps. If p is even moderately large, say at least 2^{100}, then this will take way too many steps for even modern computers.

Remark 2.3. In some ways exponentiation and logarithms have different properties when we're working in $\mathbb{F}_p$ instead of $\mathbb{R}$ or $\mathbb{C}$. Even though they behave the same way *algebraically* (i.e. they follow the same "rules"), exponentiation modulo p appears to behaves "randomly" in its input (of course it isn't actually random, but if you were to graph the function $x \mapsto g^x \pmod{p}$ for a primitive root g, then you'll get a random-looking cloud of points).

Remark 2.4. We can talk about the DLP even if g isn't a primitive root modulo p. If it isn't, then the DLP asks us to find x such that $g^x \equiv h \pmod{p}$ if such an x exists.

In fact, we can talk about the DLP in the setting of an arbitrary group. If G is a group (written multiplicatively), then the DLP for G asks us to find, for any two given g and h in G, an integer satisfying

$$\underbrace{g \cdot g \cdot \cdots \cdot g}_{x \text{ times}} = h.$$

2.3 Diffie-Hellman Key Exchange

Recall that with a symmetric cipher, Alice and Bob need to both have a key k that they use to encrypt or decrypt messages. Even if the cipher that they use is extremely resistant to any attack you can think of, if they exchange the key in an insecure way, then their communications are as good as compromised. Whitfield Diffie and Martin Hellman released a paper in 1976 showing a way of doing this with modular exponentiation, but it's since been claimed that the British GCHQ knew of this idea since the sixties and kept it secret.

The following protocol allows for Alice and Bob to establish a shared secret even in the presence of an adversary, Eve. The idea is that they can then use this secret as a key (or somehow use it to make a key) for a symmetric cipher.

1. (Public) Alice and Bob both publicly agree on a large (around 2^{1000}) prime p and an integer g relatively prime to p. Usually, g is a primitive root modulo p, but this isn't necessary.

2. (Private) Alice chooses a secret integer a and computes $A \equiv g^a \pmod{p}$. Likewise, Bob chooses a secret integer b and computes $B \equiv g^b \pmod{p}$. They perform this part completely independently of each other. Alice and Bob can complete this step efficiently with the square-and-multiply algorithm.

3. (Public) Alice sends A to Bob and Bob sends B to Alice. This happens publicly (sometimes said "in the clear"), so Eve can see this.

4. (Private) Alice uses her secret integer to compute $B^a \pmod p$ and Bob likewise computes $A^b \pmod p$. Again, this can be done efficiently with square-and-multiply.

After these steps are completed, Alice and Bob share the following value

$$\begin{aligned} A^b &\equiv (g^a)^b \pmod p \\ &\equiv (g^b)^a \pmod p \\ &\equiv B^a \pmod p. \end{aligned}$$

Eve has access to anything that was shared publicly, namely p, g, A, and B. If Eve knew either a or b, then she could simply compute B^a or A^b modulo p to compromise the shared secret. But in order for her to do this, she needs to solve either of the equations

$$g^x \equiv A \pmod p, \qquad g^y \equiv B \pmod p.$$

That is, if Eve can solve the discrete logarithm problem, then she can find the shared secret.

Example 2.5. Suppose Alice and Bob agree on the prime $p = 13$ and the integer $g = 2$. Next, say Alice randomly chooses $a = 7$ and Bob randomly (and independently) chooses $b = 9$. Alice computes $A \equiv g^a = 2^7 \equiv 11 \pmod{13}$ and Bob computes $B \equiv g^b = 2^9 \equiv 5 \pmod{13}$. Alice and Bob then exchange A and B in the clear and compute

$$B^a = 5^7 \equiv 8 \pmod{13}, \qquad A^b = 11^9 \equiv 8 \pmod{13}.$$

If Eve can find x such that $2^x \equiv 11 \pmod{13}$ or y such that $2^y \equiv 5 \pmod{13}$, then she too can perform one of the above computations to arrive at the shared value of 8.

If Eve can solve the DLP, then she can compromise Diffie-Hellman key exchange. Does she really have to do this though? Really, what she needs to do is solve the following problem.

Definition 2.6. Let p be prime and let $g \in \mathbb{F}_p^\times$, if for any a and b we can use $g^a \pmod p$ and $g^b \pmod p$ to compute $g^{ab} \pmod p$, then we can solve the *Diffie-Hellman (search) problem* or *DHP*.

Remark 2.7. The Diffie-Hellman problem doesn't actually require us to find either of the exponents a, b.

Clearly DHP is no harder than DLP, but it's *unknown* whether or not solving the DHP would allow one to solve the DLP as well.

2.4 The Elgamal Public Key Cryptosystem

The Diffie-Hellman protocol serves as a way for Alice and Bob to create a key to use in some other symmetric cipher. That is, it doesn't directly allow Alice and Bob to send messages to one another. The following cryptosystem, attributed to Egyptian cryptographer Taher Elgamal (1985), while not the first of its kind (that was RSA in 1978 - we'll talk about that later), has a similar flavor to Diffie-Hellman key exchange. The Elgamal cryptosystem proceeds as follows.

1. (Public) Alice or some trusted party publicly chooses a large prime p and an element $g \in \mathbb{F}_p^\times$ with large prime order. (In particular, g is *not* a primitive root. Why not?)

2. (Key generation) Alice chooses a random $1 \leq a \leq p-1$ and computes $A \equiv g^a \pmod{p}$, her *public key*. Alice then publishes A while keeping a, her *private key* secret.

3. (Encryption) Bob chooses a plaintext $m \in \mathbb{F}_p^\times$. He then chooses a random $1 \leq k \leq p-1$, his *ephemeral (temporary) key*, and computes $c_1 \equiv g^k \pmod{p}$ and $c_2 \equiv m \cdot A^k \pmod{p}$. He then sends the ciphertext (c_1, c_2) to Alice.

4. (Decryption) Alice uses her private key a to compute the plaintext in the following way.

$$(c_1^a)^{-1} \cdot c_2 \equiv (g^{ak})^{-1} \cdot m \cdot A^k \equiv g^{-ak} \cdot g^{ak} \cdot m \equiv m \pmod{p}.$$

Step 1 is interesting in its own right, but we'll talk about it more later when we discuss primality testing. If we assume that Alice can efficiently generate random numbers (which can be done, but we won't talk about it here), then she can efficiently compute $A \equiv g^a \pmod{p}$ with the square-and-multiply algorithm, so step 2 is efficiently doable. Similarly, Bob can efficiently compute $c_1 \equiv g^k \pmod{p}$ and $c_2 \equiv m \cdot A^k \pmod{p}$ in step 3. Finally, Alice can use square-and-multiply to efficiently compute c_1^a and then use, say, the Euclidean algorithm to compute its inverse, so decryption is also efficient.

What does an eavesdropper, Eve, need to do in order to compromise Elgamal? If she can recover the ephemeral key k that Bob uses to encrypt the plaintext m, then she can recover the plaintext by computing

$$c_2 \cdot (A^k)^{-1} \equiv m \cdot A^k \cdot A^{-k} \equiv m \pmod{p}.$$

If she knows k, then she can do this efficiently. If Eve can solve the discrete logarithm problem, then she can find k given the public quantities p and $c_1 \equiv g^k \pmod{p}$. Similarly, if she can recover Alice's private key a, then she can decrypt any messages sent to Alice (in a way, she becomes indistinguishable from Alice). Just like with k, if Eve can solve the discrete logarithm problem, then she can recover a from $A \equiv g^a \pmod{p}$. This proves the following proposition.

Proposition 2.8. *If Eve can solve the discrete logarithm problem then she can decrypt arbitrary Elgamal ciphertexts encrypted using arbitrary Elgamal public keys.*

Computational problems like this are often stated in terms of *oracles*. We won't define this term rigorously, but you can think of an oracle for a computational problem as a machine or entity that, when given an instance of that problem, solves it instantly. So another way of stating the above proposition is that Eve can decrypt arbitrary Elgamal ciphertexts with an oracle for the discrete logarithm problem.

How does the security of Elgamal compare to that of Diffie-Hellman. Both of them can be broken by access to a discrete logarithm oracle, but can we use one to break the other?

Proposition 2.9. *Fix a prime p and an element $g \in F_p^\times$ to use in Elgamal encryption. Suppose that Eve has access to an oracle that decrypts arbitrary Elgamal ciphertexts encrypted using arbitrary Elgamal public keys. Then she can use the oracle to solve the Diffie-Hellman problem.*

Conversely, if Eve has access to an oracle that solves the Diffie-Hellman problem, then she can decrypt arbitrary Elgamal ciphertexts.

Proof. We'll show how to use an Elgamal oracle to break Diffie-Hellman and you'll do the converse on your homework. Recall that in the Diffie-Hellman problem, Eve is given

$$A \equiv g^a \pmod{p} \qquad \text{and} \qquad B \equiv g^b \pmod{p}$$

and she needs to compute $g^{ab} \pmod p$. The tool Eve has at her disposal is an Elgamal oracle, which takes as input a prime p, a base g, a public key A and a ciphertext (c_1, c_2) and returns the quantity

$$(c_1^a)^{-1} \cdot c_2 \pmod p.$$

What inputs can Eve feed to her Elgamal oracle to solve the DHP? If she inputs the public key $A \equiv g^a \pmod p$ and the ciphertext $(B, 1)$, then the oracle will return

$$(B^a)^{-1} \cdot 1 \equiv g^{-ab} \pmod p.$$

Then she just has to invert this (say, with the Euclidean algorithm). Even if the oracle rejects inputs where the second part of the ciphertext is 1, she can still win. Eve can choose a random element $c_2 \in \mathbb{F}_p^\times$ and submit (B, c_2) to the oracle. The oracle will then return

$$(B^a)^{-1} \cdot c_2 \equiv g^{-ab} \cdot c_2 \pmod p.$$

Then Eve just has to multiply this by c_2^{-1} and invert to obtain g^{ab}. □

2.5 Basic Group Theory

Recall that we can add and multiply elements of $\mathbb{F}_p$. If we just look at the additive structure, then we have an identity ($0 + a = a$ for all a) and (additive) inverses (for all a, $a + (-a) = 0$). Similarly, if we look at $\mathbb{F}_p^\times$, we have an identity ($a \cdot 1 = a$ for all a) and (multiplicative) inverses (for all a, $a \cdot a^{-1} = 1$). These kinds of structures that come with an operation that admits an identity and inverses come up a lot in math and merit having their own definition.

Definition 2.10. A *group* consists of a set G along with a binary operation $\star : G \times G \to G$ that satisfies the following properties.

1. (Identity) There is an element $e \in G$ such that

   $$e \star a = a \star e = a$$

   for all $a \in G$.

2. (Inverses) For each $a \in A$ there is a (unique) element $a^{-1} \in G$ such that

   $$a \star a^{-1} = a^{-1} \star a = e.$$

3. (Associativity) For all a, b, c in G we have

   $$a \star (b \star c) = (a \star b) \star c.$$

Remark 2.11. We often drop the $\star$ from our notation and just write elements of G next to each other to indicate the operation (called *juxtaposition*). That is, we often write ab instead of $a \star b$ (we write the group *multiplicatively*). In this case, we sometimes write 1 for the identity element. If the group is *commutative*, i.e. $a \star b = b \star a$ for all a and b, then we sometimes write $a + b$ instead of $a \star b$ (we write the group *additively*) and 0 instead of e for the identity.

Remark 2.12. If we write our group multiplicatively, then we write

$$a^n = \underbrace{a \cdot a \cdot \cdots \cdot a}_{n \text{ times}}$$

for any nonnegative integer n If we write it additively, then

$$na = \underbrace{a + a + \cdots + a}_{n \text{ times}}.$$

Example 2.13. (a) $G = \mathbb{Z}$ becomes a group when we equip it with the operation of addition (the identity is 0 and the inverse of a is $-a$). However, $\mathbb{Z}$ is not a group when we equip it with multiplication since all integers not equal to ± 1 lack multiplicative inverses.

(b) $\mathbb{Z}/n\mathbb{Z}$ is a group when we equip it with addition modulo n. It's a group with respect to multiplication (when we throw out 0 of course) if and only if n is prime. In general, to get a group with respect to multiplication, we have to look at $(\mathbb{Z}/n\mathbb{Z})^\times$, the (equivalence classes of) integers coprime to n.

(c) The set of $n \times n$ matrices with real (or complex, or $\mathbb{F}_p$) entries and nonzero determinant is a group with respect to matrix multiplication. We denote this group by $\mathrm{GL}_n(\mathbb{R})$ (or $\mathrm{GL}_n(\mathbb{C})$ or $\mathrm{GL}_n(\mathbb{F}_p)$), the *general linear group.*

(d) If X is a set, then the set of all bijections from X to itself, denoted by S_X, is a group with respect to the operation of function composition (what is the identity? how do you invert?)

Lots of the algebra that we've built up for $\mathbb{Z}/n\mathbb{Z}$ (and lots of the cryptography) carries over to arbitrary groups. In particular, we still have the notion of an element's order.

Definition 2.14. Let G be a group and let $a \in G$. The smallest positive integer d such that $a^d = e$ is called the *order* of a. If there is no such integer, then we say a has *infinite order.*

We have a similar interplay between order and divisibility.

Proposition 2.15. *Let G be a finite group (i.e., $|G|$ is finite). Then every element of G has finite order. Moreover, if $a \in G$ has order d and if $a^k = e$, then $d \mid k$.*

Proof. We look at the powers of a:

$$a, a^1, a^2, \ldots$$

If these elements of G were all distinct, then G wouldn't be a *finite* group, so we must have repetitions here. Say $a^i = a^j$ for some $i < j$. Then we can multiply both sides by the inverse of a i times to get $a^{j-i} = e$. Since $j - i$ is positive, we've found a positive integer power of a that gives us e. Let d be the smallest such power (the order of a - this comes from the *well-ordered property* of the natural numbers).

Now suppose that $a^k = e$ for some positive k. Divide k by d with remainder to get

$$k = dq + r, \quad \text{with } 0 \leq r < d.$$

Then

$$e = a^k = a^{dq+r} = (a^d)^q \cdot a^r = 1 \cdot a^r = a^r.$$

Since d is the smallest positive power of a equal to e, we must have that $r = 0$ and $d \mid k$. □

We also have a version of Fermat's little theorem that works in any finite group.

Theorem 2.16 (Lagrange's Theorem)**.** *Let G be a finite group with n elements. Then $a^n = e$ for all $a \in G$. In particular, the order of a divides n.*

Proof. This theorem really is true for *any* finite group, but since most of the groups used in cryptography are commutative, you'll prove the theorem under the additional assumption that G is commutative in your discussion section. The proof of the more general statement isn't very complicated and it's one of the first things you learn in Math 120A. □

2.6 Order Notation and Algorithms

In cryptography we often find ourselves concerned with algorithms (like when we think about how to compute things or when we think about what an adversary needs to do in order to compromise or communications). More specifically, we're usually interested in how "good" and algorithm is, usually measured in the number of "steps" it takes to complete (may the number of times we perform a basic arithmetic operation like multiplication or addition or maybe the number of times we have to query an oracle).

Example 2.17. Consider the problem of finding your name on an alphabetized roster containing n names. One way we can do this is by reading the names off one by one until we find your name. In the worst case, your name is the last on the list and we have to read through all n names.

Another approach is to look at the name in the middle of the list (if there is an even number of names, just pick one of the two in the "middle" arbitrarily. If your name comes before this one in the alphabet, then we can safely throw out all the names that come after this middle name (and vice-versa if your name comes after it; if we happened to pick your name at this step, then we're done!). This just leaves $n/2$ names to look at. Now do the same thing - look at the name in the middle of this list then throw out everything after it if your name comes before this name (and throw out everything before if your name comes after).

How many steps does this take to run? At each step we cut the number of things we need to search through in half. We keep doing this until we find your name. At worst, we have to check T names, where T is the smallest integer such that $n/2^T < 1$. Therefore, this algorithm requires at most $\lfloor \log_2(n) \rfloor + 1$ steps to complete.

Remark 2.18. In this example, we bounded the number of steps it took to complete our task by considering the *worst case*. When counting the number of steps, we always assume we're working in the worst case (unless we specify otherwise).

The second algorithm in the above example, although slightly more complicated than the first, is way more efficient (again, in the worst case scenario). One way to intuit this is to compare the growth of the functions n and $\log_2 n$ – the former grows much more rapidly than the latter (if the list had 1000 names on it, then it would take at most ten steps to find your name!) This motivates the following framework for comparing the growth of functions.

Definition 2.19. Let f, g be functions $\mathbb{N} \to \mathbb{R}$. We say that $f(n) = O(g(n))$, read "f is big O of g" if there exist positive constants c and C such that

$$f(x) \leq Cg(x)$$

for all $x \geq c$. That is, f is *eventually* bounded by some constant multiple of g.

Example 2.20. If $f(n) = 3n^2 + 5\log n$, then $f(n) = O(n^2)$. To see this, note that $\log n$ is smaller than n for any $n \geq 1$. Likewise, $n \leq n^2$ for all $n \geq 1$, so we have

$$\begin{aligned} 3n^2 + 5\log n &\leq 3n^2 + 5n \qquad \text{if } n \geq 1 \\ &\leq 8n^2 \qquad \text{if } n \geq 1. \end{aligned}$$

The following proposition gives an easier way of checking if $f(n) = O(g(n))$.

Proposition 2.21. *If*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L < \infty,$$

then $f(n) = O(g(n))$.

Proof. By the definition of limits, we must have that

$$\left| \frac{f(n)}{g(n)} - L \right| \leq 1$$

for all n greater than some c. Consequently, when $n \geq c$, we have that

$$f(n) \leq (L+1)g(n),$$

so setting $C = L + 1$ in the definition of big O notation proves the claim. □

Let's record some basic facts and examples.

Example 2.22.

1. (We can ignore constant factors) For any function $g(n) : \mathbb{N} \to \mathbb{R}$ we have that $Kg(n) = O(g(n))$ for any constant K.
2. $\log n = O(n^\alpha)$ for any $\alpha > 0$. That is, $\log n$ grows more slowly than *any* polynomial. You can prove this with L'Hopital's rule.
3. If $f(n)$ is any polynomial, then $f(n) = O(2^n)$.
4. If $f(n)$ is a constant, then $f(n) = O(1)$.
5. If $f(n)$ is the number of steps it takes to find a name in an alphabetized roster, then $f(n) = O(\log n)$.
6. If $f(a, b)$ is the number of steps it takes to compute the greatest common divisor of a and b, then $f(a, b) = O(\log b)$, where $b \geq a$.

Remark 2.23. Usually we care about how the number of steps it takes an algorithm to complete scales with the size of its input. The size of the input is often measured in the number of bits it takes to store it (this is how computers store things). With this in mind, if the input to a problem is k bits long and it takes an algorithm $O(k^A)$ steps to complete for some constant $A \geq 0$, then we say that the algorithm runs in *polynomial time*. If it takes $O(2^{ck})$ steps for some $c \geq 0$, then we say it takes *exponential time*.

Between these two classes we have *subexponential time*, where the number of steps is $O(2^{\epsilon k})$ for every $\epsilon > 0$ (here the constants in the definition of big O notation are allowed to depend on ϵ). Generally speaking we consider polynomial-time algorithms to be efficient (they usually run decently well on computers if the degree of the polynomial isn't too big) and exponential time algorithms to be inefficient.

Example 2.24. The brute-force algorithm for solving the discrete logarithm problem in $\mathbb{F}_p^\times$ requires checking at most $p-1$ different values of g^x. Since p takes $k \approx \log_2 p$ bits to store, this algorithm runs in exponential time. We'll explore better algorithms later.

Example 2.25. In some groups the discrete logarithm problem is extremely easy. For example, if we consider the DLP in the *additive* group $\mathbb{F}_p$, then we want to find x such that $x \cdot g \equiv h \pmod{p}$ for fixed g and h. But we can do this by just computing $g^{-1} \pmod{p}$ and multiplying, which takes $O(\log p)$ steps with the Euclidean algorithm.

Finding groups where the discrete logarithm problem is hard is important in cryptography.

2.7 A collision algorithm for the DLP

Recall that the discrete logarithm problem (DLP) in a group G asks us to find an integer x such that

$$g^x = h$$

for some fixed g and h in G, if such an x exists. If G has N elements, then $g^N = e$ by Lagrange's theorem. We can then solve the DLP by simply trying all powers of g,

$$g,\ g^2,\ g^3, \ldots, g^N$$

until we get h (or we never get h and the DLP has no solution). This takes $O(N)$ steps and requires $O(1)$ storage since we only need to store one power of g at a time when performing this algorithm.

However, an element of G takes $b := \log N$ bits to store or specify, so this algorithm runs in $O(2^b)$ steps – exponential in the size of the input.

Let's illustrate a new algorithm with an example. Let's just say that G has $N = 100$ elements. Now consider the lists

$$A = \{e, g^1, g^2, \ldots, g^9\}, \qquad B = \{h, h \cdot g^{-10}, h \cdot g^{-20}, \ldots, h \cdot g^{-90}\}.$$

If these lists have a common element, then

$$g^a = h \cdot g^{-b \cdot 10} \implies g^{b \cdot 10 + a} = h,$$

and we've found $\log_g(h)$. If the DLP has a solution for this g and h, then there must be a match between these two lists. If x is the solution, write $x = 10q + r$ where $q, r < 10$. Then g^r is in A and $h \cdot g^{-10q}$ is in B.

Remark 2.26. An important part of this algorithm is finding a match between two lists. If the lists A and B have t elements each and they have a match between them, then we can find it in $O(t \log t)$ steps. This comes from the fact that the lists can both be sorted in $O(t \log t)$ steps (say, by using merge sort). Then go through each of the t elements of A and take at most $O(\log t)$ steps to find it in the sorted list B.

In general, suppose we want to solve $g^x = h$ in the group G, which has N elements. First let $n = 1 + \lfloor \sqrt{N} \rfloor$. Then do the following.

1. Create the two lists

$$A = \{e, g, g^2, \ldots, g^n\}, \qquad B = \{h, h \cdot g^{-n}, h \cdot g^{-2n}, \ldots, h \cdot g^{-n^2}\}.$$

2. Sort the lists A and B

3. Find a match between the lists.

4. The match is $g^i = h \cdot g^{-jn}$. Return $x = i + jn$ as a solution to $g^x = h$.

How many steps does this algorithm take?

1. The elements of A and B can both be generated by just repeatedly multiplying by a single element (by g in the list A and by g^{-n} in list B), so $O(n) = O(\sqrt{N})$ steps for step 1.

2. We've already remarked that sorting the lists A and B takes $O(n \log n) = O(\sqrt{N} \log N)$ steps.

3. Once the lists are sorted, it takes $O(\log n) = O(\log N)$ steps per element of A to see if it has a match in B, so $O(n \log n) = O(\sqrt{N} \log N)$ steps here.

Thus, the total number of steps is

$$O(\sqrt{N} + \sqrt{N} \log N + \sqrt{N} \log N) = O(\sqrt{N} \log N).$$

If the DLP has a solution, x, then the two lists must have a match. Write $x = nq + r$ with $0 \leq r < n$ and $q \geq 0$. Then g^r is in list A and $h \cdot g^{-nq}$ is on list B so long as $q \leq n$. If $q > n$, then we would have $qn > n^2 > N$. But $r \geq 0$, so we would have $x = nq + r > N$, which contradicts the fact that, if a solution exists, it must be at most N.

This algorithm is sometimes called *baby step giant step*. The elements of A are the "baby steps" and those of B are the "giant steps".

Remark 2.27. While this algorithm is definitely better than brute-force – $O(\sqrt{N} \log N)$ instead of $O(N)$, it still runs in time that is exponential in the size of the input. Indeed, since and element of N takes $b = \log N$ bits to specify, this algorithm runs in time $O(2^{b/2} \cdot b)$.

2.8 The Chinese Remainder Theorem

Consider the discrete logarithm problem in $\mathbb{F}_p^\times$. That is, we want to find a solution x to the equation

$$g^x \equiv h \pmod{p}.$$

By Fermat's little theorem, the solution to the above equation, if it exists, is defined modulo $p - 1$. Now a big idea in computer science is to take a problem and try to decompose it into smaller problems that are (hopefully) easier to solve. Which smaller problems should we break the DLP into? Well if we can factor $p - 1$ into smaller primes,

$$p - 1 = q_1^{e_1} q_2^{e_2} \cdots q_t^{e_t},$$

then maybe we can find what x is modulo each $q_i^{e_i}$ first and then somehow weave these solutions together into a solution modulo $p - 1$. Let's illustrate this with a simple example.

Example 2.28. Let's find an integer x that solves both of the congruences

$$x \equiv 2 \pmod{5} \qquad \text{and} \qquad x \equiv 3 \pmod{7}.$$

The full set of integer solutions to the first equation is

$$x = 2 + 5k, \quad k \in \mathbb{Z}. \tag{5}$$

If we substitute this into the second congruence we get

$$2 + 5k \equiv 3 \pmod 7 \implies 5k \equiv 1 \pmod 7.$$

Now we simply multiply both sides by the inverse of 5 modulo 7. This can indeed be done since $\gcd(5, 7) = 1$. We can even do it efficiently using the Euclidean algorithm (or just guess-and-check here since the numbers are small). In any case, the inverse of 5 modulo 7 is 3, so we have

$$k \equiv 3 \pmod 7.$$

That is, $k = 3 + 7k'$ for any integer k'.

If we substitute this into (5), we see that

$$2 + 5(3 + 7k) = 17 + 35k'$$

is a full set of solutions to both congruences. In other words, the solution is $x \equiv 17 \pmod{35}$.

This example gives an algorithm for how to prove the following theorem.

Theorem 2.29 (Chinese Remainder Theorem)**.** *Let $m_1, m_2, \ldots, m_k$ be a collection of pairwise coprime integers, i.e.*

$$\gcd(m_i, m_j) = 1 \quad \textit{for } i \neq j.$$

Let $a_1, a_2, \ldots, a_k$ be arbitrary integers. Then the system of simultaneous congruences

$$x \equiv a_1 \pmod{m_1},\ x \equiv a_2 \pmod{m_2}, \quad \ldots, \quad x \equiv a_k \pmod{m_k}$$

has a unique solution modulo $m_1 m_2 \ldots m_k$.

Proof. We explicitly construct the solution in a neat and algorithmic way. Suppose we already have an integer solution $x = c_i$ the the first i congruences.

$$x \equiv a_1 \pmod{m_1},\ x \equiv a_2 \pmod{m_2}, \quad \ldots, \quad x \equiv a_i \pmod{m_i}$$

For example, $c_1 = a_1$ is a solution to the first congruence. Now let's show how to extend this solution to solve one more congruence. Look at the integers of the form

$$x = c_1 + m_1 m_2 \cdots m_i y$$

as y ranges over the integers. If we reduce such an x modulo m_j for any $j \leq i$, we obtain another solution to the first i congruences, so we just need to pick y so that $x \equiv a_{i+1} \pmod{m_{i+1}}$. That is, we need to solve the equation

$$c_i + m_1 m_2 \cdots m_i y \equiv a_{i+1} \pmod{m_{i+1}}.$$

But we can do this by just moving the c_i over and inverting $m_1 \cdots m_i$ modulo m_{i+1}. We can indeed do this because the m_j's are all pairwise coprime, so

$$\gcd(m_1 m_2 \cdots m_i, m_{i+1}) = 1.$$

You'll prove the uniqueness of this solution on your homework. □

2.9 The Pohlig-Hellman Algorithm

We hope to find an easier way to solve the discrete logarithm problem (find x such that $g^x \equiv h \pmod{p}$). Our idea was to factor $p-1 = q_1^{e_1} \cdots q_t^{e_t}$ and then somehow find $x \pmod{p_i^{e_i}}$ for each i. The Chinese remainder theorem then gives us a way to (efficiently) weave these solutions together into $x \pmod{p-1}$. This is essentially the idea behind the Pohlig-Hellman algorithm.

Theorem 2.30. *Let G be a group and suppose we have an algorithm to solve the discrete logarithm problem in G for any element whose order is a prime power. That is, if $g \in G$ has order q^e, suppose we can solve $g^x = h$ in $O(S_{q^e})$ steps.*

Then if $g \in G$ has order N, where N has prime factorization

$$N = q_1^{e_1} q_2^{e_2} \cdots q_t^{e_t},$$

the discrete logarithm problem $g^x = h$ can be solved in

$$O\left(\sum_{i=1}^{t} S_{q_i^{e_i}} + \log N\right)$$

steps with the following procedure.

1. *For each $1 \le i \le t$, let*
$$g_i = g^{N/q_i^{e_i}} \qquad \text{and} \qquad h_i = h^{N/q_i^{e_i}}.$$
Solve the discrete logarithm problem
$$g_i^y = h_i$$
and let y_i be a solution.

2. *Use the Chinese remainder theorem to solve*
$$x \equiv y_1 \pmod{q_1^{e_1}}, \ \ldots, \ x \equiv y_t \pmod{q_t^{e_t}}. \tag{6}$$

Proof. Since g has order N, $g_i = g^{N/q_i^{e_i}}$ has order $q_i^{e_i}$ (why?). Consequently, the first step of our algorithm takes $O(\sum S_{q_i^{e_i}})$ steps. The Chinese remainder theorem step consists of just basic addition and inversion modulo $q_i^{e_i}$. Since inversion can be done in $O(\log q_i^{e_i}) = O(\log N)$ steps, the entirety of step 2 takes just $O(t \log N) = O(\log N)$ steps.

Since x satisfies the simultaneous congruences (6), we can write

$$x = y_i + z_i q_i^{e_i}$$

for some integer z_i. We then have

$$\begin{aligned}
(g^x)^{N/q_e^{e_i}} &= (g^{y_i + z_i q_i^{e_i}})^{N/q_i^{e_i}} \\
&= (g^{N/q_i^{e_i}})^{y_i} \cdot g^{N z_i} \\
&= g_i^{y_i} \\
&= h_i \\
&= h^{N/q_i^{e_i}}.
\end{aligned}$$

The third line follows from the fact that $a^N = e$ for any $a \in G$ by Lagrange's theorem. If we take the logarithm to the base g of both sides of this, we have

$$\frac{N}{q_i^{e_i}} x \equiv \frac{N}{q_i^{e_i}} \log_g(h) \pmod{N}. \tag{7}$$

Since the integers $N/q_1^{e_1}, \ldots, N/q_t^{e_t}$ are pairwise coprime, we can find integers $c_1, \ldots, c_t$ such that

$$\frac{N}{q_1^{e_1}} c_1 + \cdots + \frac{N}{q_t^{e_t}} c_t = 1.$$

If we multiply both sides of (7) by c_i and sum over i, then we see that

$$\begin{aligned} x &\equiv \sum_{i=1}^{t} x \cdot \frac{N}{q_i^{e_i}} c_i \\ &\equiv \sum_{i=1}^{t} \log_g(h) \cdot \frac{N}{q_i^{e_i}} c_i \\ &\equiv h \pmod{N}. \end{aligned}$$

□

We've reduced the discrete logarithm problem to the case where the base has prime power order. Now we show how to reduce the problem even further to just a base with prime order.

Proposition 2.31. *Let G be a group, q a prime and suppose we have an algorithm that takes S_q steps to solve the DLP $g^x = h$ in G whenever g has order q. Then if g has order q^e, we can solve the discrete log problem $g^x = h$ in $O(eS_q)$ steps.*

Proof. The idea is to write the unknown exponent x in base q.

$$x = x_0 + x_1 q + x_2 q^2 + \cdots + x_{e-1} q^{e-1} \quad \text{with } 0 \le x_i < q.$$

We have

$$\begin{aligned} h^{q^{e-1}} &= (g^x)^{q^{e-1}} \\ &= \left(g^{x_0 + x_1 q + \cdots + x_{e-1} q^{e-1}}\right)^{q^{e-1}} \\ &= (g^{q^{e-1}})^{x_0} \cdot \left(g^{q^e}\right)^{x_1 + x_2 q + \cdots + x_{e-1} q^{e-2}} \\ &= \left(g^{q^{e-1}}\right)^{x_0}. \end{aligned}$$

The last line follows since $g^{q^e} = 1$. Since $g^{q^{e-1}}$ has order q, this is a problem we can solve for x_0 in $O(S_q)$ steps. We essentially repeat this to find x_2. Write

$$\begin{aligned} h^{q^{e-2}} &= (g^x)^{q^{e-2}} \\ &= \left(g^{x_0 + x_1 q + \cdots + x_{e-1} q^{e-1}}\right)^{q^{e-2}} \\ &= g^{x_0 q^{e-2}} \cdot \left(g^{q^{e-1}}\right)^{x_1} \cdot \left(g^{q^e}\right)^{x_2 + x_3 q + \cdots + x_{e-1} q^{e-3}} \\ &= g^{x_0 q^{e-2}} \cdot \left(g^{q^{e-1}}\right)^{x_1}. \end{aligned}$$

But we know x_0, so we can move it over to the LHS to get

$$(h \cdot g^{-x_0})^{q^{e-2}} = (g^{q^{e-1}})^{x_1},$$

another DLP with base of order q. We then solve this problem in $O(S_q)$ steps to find x_1. In general, to get x_i, we solve the problem

$$(g^{q^{e-1}})^{x_i} = \left(h \cdot g^{-x_0 - x_1 q - \cdots - x_{i-1} q^{i-1}}\right)^{q^{e-i-1}}.$$

In total, this takes $O(eS_q)$ steps. □

The moral of the story is that if $p-1$ factors into many small primes, then then the Pohlig-Hellman algorithm gives a viable attack on the DLP.

3 Integer Factorization and RSA

3.1 Euler's Formula and Roots Modulo pq

So far we've introduced a key-exchange protocol and a cryptosystem (Diffie-Hellmand and ElGamal, respectively) whose security relies on the same one-way operation. It's easy to exponentiate modulo n, but it's (presumably) hard to take logarithms. In this chapter we'll explore a different hard problem and build a cryptosystem on top of it.

Recall that Fermat's little theorem tells us that $a^{p-1} \equiv 1 \pmod{p}$ for any prime p and any a coprime to p. Can we generalize this theorem to composite moduli? We definitely need the base to be relatively prime to m – if $a^k \equiv 1 \pmod{m}$, then

$$a^k - \ell m = 1$$

for some integer ℓ, so $\gcd(a, m)$ would have to divide 1. It looks like the right setting for our generalized Fermat's little theorem is $(\mathbb{Z}/m\mathbb{Z})^\times$. Well if we set $m = 15$ and $a = 2$, we can easily check, say with square-and-multiply, that

$$a^{m-1} = 2^{14} = 2^2 \cdot 2^4 \cdot 2^8 \equiv 4 \cdot 1 \cdot 1 \equiv 4 \not\equiv 1 \pmod{15}.$$

So even if we take a base that's relatively prime to m, it isn't always the case that $a^{m-1} \equiv 1 \pmod{m}$.

It looks like we need to change the exponent to get an analog of Fermat's little theorem for composite moduli. Lagrange's theorem gives us the right exponent.

Definition 3.1. For any positive integer m, we define *Euler's totient function* $\phi(m)$ (sometimes written $\varphi(m)$) to be

$$\phi(m) = |\{a : 1 \le a \le m-1 \text{ and } \gcd(a, m) = 1\}| = |(\mathbb{Z}/m\mathbb{Z})^\times|.$$

We can apply Lagrange's theorem to the commutative group $(\mathbb{Z}/m\mathbb{Z})^\times$ to get the following theorem.

Theorem 3.2 (Euler's theorem)**.** *For any positive integer m and any a relatively prime to m,*

$$a^{\phi(m)} \equiv 1 \pmod{m}.$$

Remark 3.3. In the case where $m = pq$ is a product of two primes, $\phi(m) = (p-1)(q-1)$. To see this, note $\gcd(m, a) > 1$ for $1 \leq a \leq m-1$ if and only if a is a multiple of either p or q. There are p multiples of q and q multiples of p and they only coincide at pq since p and q are distinct primes. So we have

$$a^{(p-1)(q-1)} \equiv 1 \pmod{pq}$$

for all a relatively prime to pq.

Remark 3.4. We can actually improve the exponent in the above remark a bit. Let $g = \gcd(p-1, q-1)$, which is always at least 2 when p and q are odd primes. By Fermat's little theorem we have for any a relatively prime to pq,

$$\begin{aligned} a^{(p-1)(q-1)/g} &= \left(a^{p-1}\right)^{(q-1)/g} \\ &\equiv 1^{(q-1)/g} \pmod{p}. \end{aligned}$$

Note that the RHS of the first line is okay since $(q-1)/g$ is an integer. If we exchange the roles of p and q, we see that $a^{(p-1)(q-1)/g} \equiv 1 \pmod{q}$ as well. We then have that $a^{(p-1)(q-1)/g} - 1$ is divisible by both p and q, so it must be divisible by pq and we have

$$a^{(p-1)(q-1)/g} \equiv 1 \pmod{pq}.$$

Diffie-Hellman works because it's easy to compute $g^a \pmod{p}$ if we know g and a, but it's hard to get a back from it. What if we reverse the roles of the secret and public parts? It's definitely easy to compute $m^e \pmod{N}$ if we know m and e, but is it easy to get m back if we know e?

Well if p is prime it is.

Proposition 3.5. *Let p be a prime and let $e \geq 1$ be an integer satisfying $\gcd(e, p-1) = 1$. If d is the inverse of e modulo $p-1$ (which can be easily computed from e with the Euclidean algorithm), then the congruence*

$$x^e \equiv c \pmod{p}$$

has the unique solution $x \equiv c^d \pmod{p}$.

Proof. If $c \equiv 0 \pmod{p}$, then $x \equiv 0 \pmod{p}$ is the only solution, so we assume $c \not\equiv 0 \pmod{p}$. Write $de = 1 + k(p-1)$ for some integer k. Then by Fermat's little theorem we have

$$\begin{aligned} (c^d)^e &\equiv c^{1+k(p-1)} \pmod{p} \\ &\equiv c \cdot (c^{p-1})^k \pmod{p} \\ &\equiv c \pmod{p}. \end{aligned}$$

So $x \equiv c^d \pmod{p}$ is a solution to $x^e \equiv c \pmod{p}$.

As for uniqueness, suppose that x_1 and x_2 are two solutions. Then we have

$$\begin{aligned} x_1 &\equiv x_1^{de} \pmod{p} \\ &\equiv (x_1^e)^d \pmod{p} \\ &\equiv c^d \pmod{p} \\ &\equiv (x_2^e)^d \pmod{p} \\ &\equiv x_2 \pmod{p}. \end{aligned}$$

□

Example 3.6. Let's solve

$$x^5 \equiv 9 \pmod{17}.$$

First we find the inverse of 5 modulo $17 - 1 = 16$, which is 13. Then we raise both sides to the power 13 to obtain the unique solution

$$(x^5)^{13} \equiv 9^{13} \pmod{17} \iff x \equiv 8 \pmod{17}.$$

It turns out that Proposition 3.5 carries over just fine to composite moduli, but we'll see that there's some subtlety that gets introduced when we actually try to solve the congruence.

Proposition 3.7. *Let N be a positive integer and let $e \geq 1$ satisfy*

$$\gcd\big(e, \phi(N)\big) = 1.$$

If d is the inverse of e modulo $\phi(N)$, then the congruence

$$x^e \equiv c \pmod{N}$$

has the unique solution $x \equiv c^d \pmod{N}$.

Proof. The proof is nearly identical to that of Proposition 3.5. There's a slight issue if $\gcd(c, N) \neq 1$, but you'll take care of this on your next homework assignment. □

Remark 3.8. In the case where $N = pq$ is the product of distinct odd primes, Remark 3.4 offers us a potential speedup. Instead of computing the inverse of e modulo $\phi(pq) = (p-1)(q-1)$, we can instead let d be the inverse of e modulo $(p-1)(q-1)/g$, where $g = \gcd(p-1, q-1)$. In this case we have $de = 1 + k(p-1)(q-1)/g$ and

$$(c^d)^e \equiv c^{1+k(p-1)(q-1)/g} \equiv c \cdot (c^{(p-1)(q-1)/g})^k \equiv c \pmod{pq}.$$

Example 3.9. Let's solve

$$x^7 \equiv 51 \pmod{77}.$$

Since $77 = 7 \cdot 11$ is a product of two distinct primes, we look for the inverse of 7 modulo $(7-1)(11-1)/\gcd(7-1, 11-1) = 30$. This inverse is 13, so we have

$$x \equiv 51^{13} \equiv 2 \pmod{77}.$$

3.2 RSA

When we solved the congruence in the previous example, we used the factorization of 77 in order to get the right exponent. If the numbers involved are large, it looks like we'd need to be able to factor large numbers to solve congruences of this type in this way. If this is hard to do, then we can build a cryptosystem out of it. This gives us the RSA cryptosystem, due to Ron Rivest, Adi Shamir, and Leonard Adleman in 1977.

1. (Key generation) Bob chooses secret primes p and q and an integer e with $\gcd(e, (p-1)(q-1)) = 1$. He publishes $N = pq$ and e, his *public key*. He also computes d such that $ed \equiv 1 \pmod{(p-1)(q-1)}$. We call N the *modulus*, e the *encryption exponent* and d the *decryption exponent*.

2. (Encryption) Alice chooses a message $1 \leq m \leq N$. She computes $c \equiv m^e \pmod{N}$ and sends c to Bob.

3. (Decryption) Bob computes
$$c^d \equiv (m^e)^d \equiv m \pmod{N}.$$

Example 3.10. Bob chooses secret primes $p = 101$ and $q = 163$. He randomly chooses integers between 1 and $(p-1)(q-1) = 16200$ until he gets one that's relatively prime to 16200. He settles on $e = 133$ and publishes it along with $N = pq = 16463$. Finally, he uses the Euclidean algorithm to compute the inverse of e modulo 16200 and gets $d = 10597$

Alice wants to send Bob the message $m = 9876$. She computes

$$c \equiv m^e = 9876^{133} \equiv 7025 \pmod{N = 16463},$$

and sends this ciphertext to Bob.

To decrypt, Bob computes

$$c^d = 7025^{10597} \equiv 9876 = m \pmod{N = 16463}.$$

An eavesdropper, Eve, sees the modulus N, the encryption exponent e, and the ciphertext $c \equiv m^e \pmod{N}$. Eve could decrypt the ciphertext if she could somehow get the decryption exponent d. She could invert e modulo $(p-1)(q-1)$, but it's unclear how she can do this without knowing p or q. It looks like the security of RSA relies on how hard a time Eve has factoring large integers like N.

3.3 Primality Testing

If Bob wants to set up a shared key with Alice using Diffie-Hellman or RSA (in practice, RSA is used for key-exchange), it looks like he needs to generate large prime numbers. Consider this strategy.

1. Randomly generate a large number N.

2. Check to see if N is prime.

Is this a viable strategy? Generating large random numbers is easy: if we want to randomly generate an n-bit number, we can just flip a coin for each of its bits (and we can modify this slightly to generate numbers in a particular range). What about checking to see if our randomly generated N is prime? If this is hard or inefficient to do, then it would be hard for us to set up Diffie-Hellman or RSA. *Even if it's easy for us to test N for primality*, if prime numbers are extremely "rare", then it might take us way too long to actually find a prime number this way.

The content of the next few sections is addressing these issues: primes *are not* too rare for this strategy to work and it *is* easy enough for us to check to see if a number is prime.

Let's focus on step 2 for now – testing N for primality. Probably the simplest way is to simply check to see if N is divisible by every $a < N$. If N is an n-bit number, this is pretty bad – it takes $O(2^n)$ steps. We can speed up this "trial division" algorithm by only checking numbers up to $\sqrt{N}$. This works since if $N = ab$, one of a or b must be at most $\sqrt{N}$. This still runs in exponential time at $O(2^{n/2})$ steps.

Let's think about Fermat's little theorem for a minute.

Proposition 3.11. *If p is prime then*

$$a^p \equiv a \pmod{p}.$$

Proof. Note that this is slightly different from Fermat's little theorem, but it immediately follows from it. If $p \mid a$, then it's clearly true, and if $p \nmid a$, then we simply multiply both sides of $a^{p-1} \equiv 1 \pmod{p}$ by a. □

Conversely, if $a^N \not\equiv a \pmod{N}$ for some a, then N is *not* prime.

Example 3.12. We can quickly verify with square-and-multiply that

$$2^{121} \equiv 112 \pmod{121},$$

so 121 is not prime. We say that 2 is a *witness* for the compositeness of n.

Example 3.13. It would take us a while to verify this by hand, but it's true that

$$a^{561} \equiv a \pmod{561}$$

for all a. However, $561 = 3 \cdot 11 \cdot 17$. Such composite numbers are called *Carmichael numbers*.

The last example shows that Fermat's little theorem only goes "one way". That is, just because $a^N \equiv a \pmod{N}$ for all N, we *cannot* conclude that N is prime. Fermat's little theorem might not be enough to make a good primality test, but the following proposition will help us strengthen it.

Proposition 3.14. *Let p be an odd prime and write*

$$p - 1 = 2^k q \quad \textit{with } q \textit{ odd.}$$

Let a be any number not divisible by p. Then one of the following must hold

(i) $a^q \equiv 1 \pmod{p}$.

(ii) One of $a^q, a^{2q}, a^{4q}, \ldots, a^{2^{k-1}1}$ *is congruent to* -1 *modulo* p.

Proof. If $a^q \equiv 1 \pmod{p}$, then condition (i) holds and we're done. In any case, consider the list

$$a^q,\ a^{2q},\ a^{4q}, \ldots,\ a^{2^{k-1}q},\ a^{2^k q}. \tag{8}$$

The last number on this list is a^{p-1}, which is congruent to 1 modulo p by Fermat's little theorem. Furthermore, each item on this list is the square of the one that comes immediately before it. So eventually, some number on this list squares to 1. But the only elements of $\mathbb{F}_p$ that square to 1 are 1 and -1, so if the first item on this list isn't 1, there must be a -1 in this list. □

We can use this proposition to build a better version of the Fermat's little theorem primality test. Specifically, suppose n is odd and $n - 1 = 2^k q$ where q is odd. If there's an integer a with $\gcd(a, n) = 1$ and

(i) $a^q \not\equiv 1 \pmod{n}$.

(ii) $a^{2^i q} \not\equiv -1 \pmod{n}$ for all $i = 0, 1, 2, \ldots, k-1$,

then n must be composite and we say a is a *Miller-Rabin witness* for the compositeness of n.

This motivates the Miller-Rabin primality test, which test n for primality using the potential witness a. Specifically, n must be composite if conditions (i) and (ii) *both* fail. However, if one these conditions holds, we learn nothing (the proposition only goes one way).

1. If n is even or $1 < \gcd(a, n) < n$, return **Composite**.
2. Write $n - 1 = 2^k q$ where q is odd.
3. Set $a_0 \equiv a^q \pmod{n}$.
4. If $a_0 \equiv 1 \pmod{n}$, return **Inconclusive**.
5. For each $i = 0, 1, \ldots, k-1$ do the following
   (a) If $a_i \equiv -1 \pmod{n}$, return **Inconclusive**.
   (b) Set $a_{i+1} \equiv a_i^2 \pmod{n}$.
6. Return **Composite**.

It's clear that step 1 is a quick and easy test for compositeness, since we can efficiently compute GCD's. Step 2 is also efficiently doable since we aren't fully factoring $n - 1$. We're just peeling off all the 2's. Step 3 is efficiently doable since we can just square-and-multiply.

If the condition in step 4 is true, then it just tells us that condition (i) in the previous proposition holds. This holds if n is prime, but it could also hold if n is not prime, so we don't learn anything. However, if the condition doesn't hold, then n might be prime. We'd just need to check that condition (ii) from the previous proposition also fails.

Similarly, if the condition in step 5(a) holds, then this just tells us that condition (ii) in the previous proposition holds, which would happen if n is prime. Step 5(b) just has us move to the next number in the list (8).

Finally, if we reach step 6, then conditions (i) and (ii) of the previous proposition both fail, so n cannot be prime and a is a Miller-Rabin witness.

Remark 3.15. What is the runtime of the Miller-Rabin test? Step 1 is just a GCD computation, which takes $O(\log n)$ steps. Step 2 also takes $O(\log n)$ steps – 2 goes into $n-1$ at most $\log n$ times. Step 3 and each iteration of step 5(b) take $O(\log n)$ steps – square and multiply. Finally, $k \leq \log n$, so in total, the Miller Rabin test takes $O(\log n)$ steps, so it's a reasonably efficient algorithm.

In practice, we would like to pick a bunch of random values of a and run the Miller-Rabin test on n and a. Now this is only a good idea if there are no Carmichael-like numbers for the Miller-Rabin test – composite numbers that have no Miller-Rabin witnesses. It happens to be the case that every composite number has many Miller-Rabin witnesses.

Theorem 3.16. *Let n be a composite number. Then at least $3/4$ of the numbers a between 1 and $n-1$ are Miller-Rabin witnesses for n.*

This gives us a good recipe to check if a given number n is prime. Just choose, say, 10 random numbers a between 2 and n. If n is composite then each time we choose, we have a $3/4$ chance of picking a Miller-Rabin witness. Therefore, if n is composite, there is only a 4^{-10} chance that each run of the Miller-Rabin test comes up inconclusive, so it's reasonable to guess that n is prime.

Example 3.17. Let's check to see if $n = 3913$ is prime using the Miller-Rabin test. This number is small enough for trial division to be a good enough algorithm, but we'll use Miller-Rabin anyway.

$$n - 1 = 2^3 \cdot 489.$$

Let's test with $a = 2$. First we compute

$$2^{489} \equiv 3333 \pmod{2913}.$$

Since this isn't 1, we keep going.

$$2^{2\cdot 489} \equiv 3795 \pmod{3913}$$
$$2^{4\cdot 489} \equiv 2185 \pmod{2913}.$$

Since none of these is -1, we conclude that 3913 is composite.

Remark 3.18. It's important to note that the Miller-Rabin test *cannot* tell you that a number is prime, but it *can* tell you that a number is composite. If you run the test and find a Miller-Rabin witness, you know that n is composite. However, if you run the test 10 times, then if n is composite, you could have just gotten really unlucky and failed to draw a witness. In practice, since the Miller-Rabin test is so efficient, we can run it a large number of times and end up pretty certain that a number is prime if the test comes up inconclusive each time. This turns out to be good enough in applications

3.3.1 The Distribution of the Set of Primes

Remember that our goal was to come up with a good way of generating large primes. The previous section showed that we could efficiently check if a randomly chosen number is prime, but what if it takes us a really long time to actually choose a prime? To address this, let's count primes.

Definition 3.19. For any number x, let

$$\pi(x) = \{p : p \text{ is prime and } 2 \leq p \leq x\}.$$

We call π the prime-counting function.

For example, $\pi(11) = 5$ since there are five primes between 2 and 11: 2, 3, 5, 7, 11. The following theorem, whose proof we don't go into here, estimates $\pi(x)$.

Theorem 3.20.

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1.$$

Roughly speaking, this theorem says that if N is large and you randomly pick an integer between 1 and N, then there is roughly a $1/\ln N$ chance that you picked a prime.

Example 3.21. The primes we use in some real-world cryptography are around 1024 bits in length. How many primes p satisfy $2^{1023} < p < p^{1024}$? Well the prime number theorem says this is

$$\pi(2^{1024}) - \pi(2^{1023}) \approx \frac{2^{1024}}{\ln 2^{1024}} - \frac{2^{1023}}{\ln 2^{1023}} \approx 2^{1013.53}.$$

So there are a lot of primes here. In particular, if we randomly pick an integer in this range, then we have roughly a

$$2^{1013.53}/2^{1023} \approx 0.14\%$$

chance of selecting a prime. On average it should then take us around 700 guesses before we find a prime. You'll see how to improve this a bit on your homework.

It might be unsatisfying that the Miller-Rabin test is probabilistic, i.e. there is a chance that it "misses" that its input is composite. However, the test runs so quickly that we can be very confident in its results in a very short amount of time. If we're being honest, the Miller-Rabin test is deterministic, but we'd have to check at most $1/4$ of the numbers between 1 and n, and this would run in exponential time.

If the generalized Riemann hypothesis holds, then we can turn the Miller-Rabin test into a *deterministic* polynomial time algorithm.

Theorem 3.22 (G.L. Miller - 1975, E. Bach - 1990)**.** *If a generalized version of the Riemann hypothesis is true, then every composite n has a Miller-Rabin witness a for its compositeness satisfying*

$$a \leq 2(\ln n)^2.$$

In this setting, we'd only have to check $O((\log n)^2)$ potential witnesses instead of $O(n)$. More recently, unconditional polynomial time algorithms have been found.

Theorem 3.23 (M. Agrawal, N. Kayal, N. Saxena - 2002)**.** *For every $\epsilon > 0$, there is an algorithm that determines whether a given number is prime in $O((\ln n)^{6+\epsilon})$ steps.*

3.4 Pollard's $p-1$ Factorization Algorithm

RSA requires us to create large primes to use in our public keys, and the last section shows us how to do this efficiently. But the security of RSA relies on an adversary having a hard time factoring large numbers. The next algorithm shows that some large integers are easier to factor than others, even if they're products of two primes like an RSA modulus. The upshot is that it isn't good enough for us to just generate large primes and multiply them together to make an RSA modulus.

Suppose that $N = pq$ where p and q are large unknown primes, like in RSA. Let's say that there's some integer L such that

$$(p-1) \mid L \text{ and } (q-1) \nmid L.$$

Remember we don't know p or q at this point, so we'll probably have to guess what this L is. Now in this case there must be integers i, j and $k \neq 0$ such that

$$\begin{aligned} L &= i(p-1) \\ L &= j(q-1) + k. \end{aligned}$$

Then by Fermat's little theorem we have

$$\begin{aligned} a^L &\equiv (a^{p-1})^i \equiv 1 \pmod{p} \\ a^L &\equiv (a^{q-1})^j \cdot a^k \equiv a^k \pmod{q} \end{aligned}$$

for any a satisfying $\gcd(a,p) = \gcd(a,q) = 1$. If a isn't a primitive root modulo q then we might have that $a^k \equiv 1 \pmod{q}$ depending on what k is, but this should be pretty rare if the numbers involved are large. In other words, it's likely that

$$\begin{aligned} p &\mid a^L - 1 \\ q &\nmid a^L - 1. \end{aligned}$$

But then we must have

$$\gcd(a^L - 1, N) = p.$$

If we've found this magic value of L, then the above GCD calculation is really easy to do and we've found a factor of N. But where should this L come from?

If it happens to be the case that $p-1$ is divisible by a bunch of small primes, then since $(p-1)$ divides L, the same must be true for L. So a way to look for L is to just multiply small primes together until we get one that works, or take $L = n!$ for some small n. Let's turn this into an algorithm.

1. Set $a_1 = 2$ or whatever number you want.
2. For each $j = 2, 3, \ldots, T$ for some T to be specified ahead of time, do the following.
   (a) Set $a_j \equiv a_{j-1}^j \pmod{N}$.
   (b) Compute $d = \gcd(a_j - 1, N)$.
   (c) If $1 < d < N$, then d is a nontrivial factor of N and we stop the algorithm

Note that on iteration j of step 2, we have $a_j \equiv a_1^{j!} \pmod{N}$.

Example 3.24. If we take $N = 168441398857$, then it takes just a few (for a computer) steps of Pollard's algorithm to factor N. It turns out that

$$\gcd(2^{53!} - 1, N) = 350437,$$

and dividing this out gives the full factorization $N = 350437 \cdot 480661$. This works because

$$p - 1 = 350436 = 2^2 \cdot 3 \cdot 19 \cdot 29 \cdot 53.$$

The moral of the story here is that when we generate large primes p and q for use in RSA, we should check to see if $p-1$ or $q-1$ are divisible by small primes.

3.5 Factorization via Difference of Squares

We can get some inspiration for factoring from basic algebra. Recall the difference of squares identity:

$$x^2 - y^2 = (x - y)(x + y).$$

If we can write N as a difference of squares, then the above identity tells us that we can factor! For example, if $N = 247$, notice that

$$247 = 256 - 9 = 16^2 - 3^2 = (16 - 3)(16 + 3) = 13 \cdot 19.$$

We can still work with this even if N isn't a difference of squares. If $kN = X^2 - Y^2$ for some integer k, then

$$kN = (X - Y)(X + Y).$$

We can then try to factor N by computing $\gcd(N, X \pm Y)$. The idea here is that the perfect squares are pretty spread out as they get larger, so it's unlikely that $N + b^2$ is going to be a perfect square when we try random values of b. If we can somehow write a multiple of N as a difference of squares, $kN = X^2 - Y^2$, then hopefully some of the factors of $X \pm Y$ are also factors of N.

Example 3.25. Say we're trying to factor 143. It happens that

$$3 \cdot 143 = 23^2 - 10^2.$$

We then compute $\gcd(143, 23 - 10)$ and see that $143 = 13 \cdot 11$.

In general, it looks like we want integers a and b such that

$$a^2 \equiv b^2 \pmod{N}$$

and one of $a \pm b$ has a nontrivial factor in common with N. How do we find such a and b? In practice we don't look for them directly. Instead, we use the following procedure.

1. (Relation Building) Find many integers a_1, a_2, $\ldots$, a_r with the property that $c_i \equiv a_i^2 \pmod{N}$ is a product of small primes.
2. (Elimination) Take a product $c_{i_1} c_{i_2} \cdots c_{i_s}$ of some of the c_i's so that every prime in the product appears to an even power. Then
$$c_{i_1} c_{i_2} \cdots c_{i_s} = b^2$$
is a perfect square.
3. (GCD Computation) Let $a = a_{i_1} \cdots a_{i_s}$ and compute $d = \gcd(N, a - b)$. Note that
$$a^2 = (a_{i_1} \cdots a_{i_s})^2 \equiv c_{i_1} \cdots c_{i_s} \equiv b^2 \pmod{N},$$
so there's a chance that d is a nontrivial factor of N.

Let's assume for now we've completed step 1, so we have integers

$$\begin{aligned} c_1 &= p_1^{e_{1,1}} p_2^{e_{1,2}} \cdots p_t^{e_{1,t}} \\ c_2 &= p_1^{e_{2,1}} p_2^{e_{2,2}} \cdots p_t^{e_{2,t}} \\ &\vdots = \vdots \\ c_r &= p_1^{e_{r,1}} p_2^{e_{r,2}} \cdots p_t^{e_{r,t}}, \end{aligned}$$

where each of the $e_{i,j}$ are nonnegative and t is the number of primes we take to be "small". Step 2 requires us to take a product of these c_i's so that we get a perfect square. We can think of this product as

$$c_1^{u_1} c_2^{u_2} \cdots c_r^{u_r},$$

where each u_i is 0 or 1 (think of a 1 as meaning "select c_i to be in the product"). Expanding the product gives

$$\prod_{i=1}^{r} c_i^{u_i} = \prod_{j=1}^{t} p_t^{\sum_{i=1}^{r} e_{i,j} u_i}.$$

This quantity is a perfect square if and only if each sum appearing in an exponent on the RHS is an even number. That is, we need the following equations to hold

$$\begin{aligned} e_{1,1}u_1 + e_{2,1}u_2 + \cdots + e_{r,1}u_r &\equiv 0 \pmod{2} \\ e_{1,2}u_1 + e_{2,2}u_2 + \cdots + e_{r,2}u_r &\equiv 0 \pmod{2} \\ &\vdots \\ e_{1,t}u_1 + e_{2,t}u_2 + \cdots + e_{r,t}u_r &\equiv 0 \pmod{2}. \end{aligned}$$

But this is really just a system of linear equations in the variables $u_1, \ldots, u_r$, where our scalars come from the field $\mathbb{F}_2$ instead of $\mathbb{R}$ or $\mathbb{C}$. Linear algebra works in exactly the same way here though: we can solve this system (efficiently!) through standard techniques like row reduction.

4 Digital Signatures

4.1 What is a Digital Signature?

Your phone probably downloads software updates every now and then. How does your phone know that the updates are actually coming from the developer and not some hacker? The answer is that the developer can "sign" the software update and your phone has a way of verifying this signature.

This scenario differs slightly from the one we've seen with RSA and ElGamal. With RSA, Bob wants to be able to securely read messages that anybody sends him. In this digital signature scenario, Bob wants to broadcast messages (or software updates in the previous example) and anybody should be able to verify that these messages are indeed coming from Bob.

Even though these scenarios sound like opposites in a way, we can use ideas from public key cryptography to approach both of them. Here's such an approach to digital signatures.

1. The author has a private *signing key*, k^s.
2. The author publishes a public *verification key*, k^v.
3. A *signing algorithm* takes a document D from the author along with their signing key k^s and outputs a signature D^s for the document.
4. A *verification algorithm* takes a document D, a signature D^s, and a verification key k^v. The algorithm returns TRUE if D^s is a signature for D associated with the signing key k^s. Otherwise, it returns FALSE.

It's important to note that the verification algorithm does *not* know what the signing key is. What properties should signing and verifying have? Suppose an attacker knows the verification key and some signatures $D_1^s, \ldots, D_n^s$ for some documents $D_1, \ldots, D_s$. Then the following are nice properties to have.

- (Resistance to total breaks) An attacker should not be able to determine the signing key.
- (Resistance to universal forgery) An attacker shouldn't be able to sign arbitrary documents.
- (Resistance to selective forgery) An attacker shouldn't be able to sign a document of their choice.
- (Resistance to existential forgery) An attacker shouldn't be able to sign some document they don't already know the signature for.

Remark 4.1. In practice, it would be far too slow to input an entire document (think of a large file like an HD movie) and some signing algorithms only accept documents of a fixed small size. Instead, we usually sign a *hash* of the document. A hash function takes in documents of arbitrary size and spits out, say, a bit string of a fixed size,

$$H : \{\text{arbitrary documents}\} \to \{0,1\}^k$$

and is hard to invert or find collisions for. Instead of signing D, we sign $H(D)$.

4.2 RSA Digital Signatures

It turns out that we can use RSA for signatures too (both uses of RSA were outlined in the original 1977 paper). It looks just like the RSA protocol we already know but in reverse.

1. (Key generation) Samantha chooses secret primes p and q. She chooses verification exponent e such that
$$\gcd(e, (p-1)(q-1)) = 1.$$
She also computes the signing exponent d such that
$$de \equiv 1 \pmod{(p-1)(q-1)},$$
and keeps it secret. She publishes $N = pq$ and e.

2. (Signing) Samantha signs document D by computing
$$S \equiv D^d \pmod{N}.$$

3. (Verification) Victor accepts the signature as valid if and only if $S^e \equiv D \pmod{N}$.

If Samantha really signed D with her signing exponent, then
$$S^e \equiv D^{de} \equiv D \pmod{N}$$
by Euler's theorem. Just like with the RSA messaging protocol, if Eve can factor N, then she can recover the signing exponent from the verification exponent.

Remark 4.2. Technically, Eve doesn't have to factor N. She just needs to be able to compute the e-th root of the document to forge its signature.

Remark 4.3. We can again get a minor speedup by computing the inverse of e modulo $(p-1)(q-1)/\gcd(p-1, q-1)$ instead.

4.3 ElGamal Signatures and DSA

We can build a signature protocol based on the discrete logarithm problem too, but it's not quite as straightforward as with RSA (but still pretty simple). Here's one based on ElGamal.

1. (Parameter creation) Samantha (or a trusted party) publishes a large prime p and a primitive root g modulo p.

2. (Key generation) Samantha chooses $1 \leq a \leq p-1$ to be her *signing key*. She publishes $A \equiv g^a \pmod{p}$, the verification key.

3. (Signing) Samantha chooses a document $D \pmod{p-1}$. She then chooses a random $1 < k < p$ such that $\gcd(k, p-1) = 1$. Finally, she computes the signature in two pieces (S_1, S_2) as follows.
$$\begin{aligned} S_1 &\equiv g^k \pmod{p} \\ S_2 &\equiv (D - aS_1)k^{-1} \pmod{p-1}. \end{aligned}$$

4. (Verification) Victor accepts the signature as valid if and only if

$$A^{S_1} S_1^{S_2} \equiv g^D \pmod{p}.$$

To see why this algorithm works, suppose Samantha signs the document D and obtains S_1 and S_2 as above. Then

$$\begin{aligned} A^{S_1} S_1^{S_2} &\equiv g^{aS_1} g^{kS_2} \\ &\equiv g^{aS_1 + k(D - aS_1)k^{-1}} \\ &\equiv g^d \pmod{p}. \end{aligned}$$

Notice that since the quantity S_2 appears in the exponent, it makes sense to consider it modulo $p-1$. It's also clear that if Eve can solve the discrete logarithm problem, then she can compute the signing key and totally break this system. Does she really need to solve the DLP though? What she really needs are integers x and y such that

$$A^x x^y \equiv g^D \pmod{p}.$$

We can definitely solve this if we can solve the DLP: take the logarithm of both sides to the base g to obtain

$$x \log_g A + y \log_g x \equiv D \pmod{p-1}.$$

Then just plug in an arbitrary x and solve this for y.

One drawback of this signature protocol is that the signature is pretty long. In practice, p is between 1000 and 2000 bits, so signatures are between 2000 and 4000 bits. We now describe an algorithm that gives much shorter signatures.

The issue is that p needs to be really large for the ElGamal algorithm to be secure against attacks like Pohlig-Hellman or index calculus (we didn't talk about this second one). Instead of working in all of $\mathbb{F}_p^\times$, we pass to a subgroup of order q, where q is a prime that's considerably smaller than p. If it happens to be the case that solving the DLP in this subgroup isn't any easier than solving it in $\mathbb{F}_p^\times$, then we can obtain shorter signatures by using elements from this subgroup instead (since elements of a smaller group will take fewer bits to specify). This is the idea behind the Digital Signature Algorithm (DSA).

1. (Parameter creation) Samantha (or a trusted party) chooses large primes p and q satisfying $p \equiv 1 \pmod{q}$ and an element g of order q modulo p. She can do this by first taking a primitive root g_1 of $\mathbb{F}_p^\times$ and setting
$$g = g_1^{(p-1)/q}.$$

2. (Key creation) Samantha chooses a secret signing key $1 \leq a < q$. She computes $A \equiv g^q \pmod{p}$ and publishes the verification key A.

3. (Signing) Samantha chooses a document $D \pmod{q}$. She chooses a random $1 < k < q$ and computes the signature

$$\begin{aligned} S_1 &\equiv (g^k \pmod{p}) \pmod{q} \\ S_2 &\equiv (D + aS_1)k^{-1} \pmod{q}. \end{aligned}$$

4. (Verification) Victor computes

$$V_1 \equiv DS_2^{-1} \pmod{q}$$
$$V_2 \equiv S_1 S_2^{-1} \pmod{q}$$

and accepts the signature as valid if and only if

$$(g^{V_1} A^{V_2} \pmod{p}) \equiv S_1 \pmod{q}.$$

In practice, p is between 1000 and 2000 bits while q is between 160 and 320 bits. The rest proceeds like the ElGamal signature protocol with some minor twists.

References

[1] Hoffstein, Jeffrey, Jill Pipher and Joseph H. Silverman. *An Introduction to Mathematical Cryptography*. Second Edition. Springer New York, NY. 2014.