Number Theory II

The man pictured above is Alan Turing, the most important figure in the history of computer science. For decades, his fascinating life story was shrouded by government secrecy, societal taboo, and even his own deceptions.

At 24 Turing wrote a paper entitled *On Computable Numbers, with an Application to the Entscheidungsproblem*. The crux of the paper was an elegant way to model a computer in mathematical terms. This was a breakthrough, because it allowed the tools of mathematics to be brought to bear on questions of computation. For example, with his model in hand, Turing immediately proved that there exist problems that no computer can solve—no matter how ingenius the programmer. Turing's paper is all the more remarkable because he wrote it in 1936, a full decade before any computer actually existed.

The word "Entscheidungsproblem" in the title refers to one of the 28 mathematical problems posed by David Hilbert in 1900 as challenges to mathematicians of the 20th century. Turing knocked that one off in the same paper. And perhaps you've heard of the "Church-Turing thesis"? Same paper. So Turing was obviously a brilliant guy who generated lots of amazing ideas. But this lecture is about one of Turing's less-amazing ideas. It involved codes. It involved number theory. And it was sort of stupid.

1 Turing's Code

Let's look back to the fall of 1937. Nazi Germany was rearming under Adolf Hitler, world-shattering war looked imminent, and— like us— Alan Turing was pondering the useful-

ness of number theory. He forsaw that preserving military secrets would be vital in the coming conflict and proposed a way *to encrypt communications using number theory*. This is an idea that has ricocheted up to our own time. Today, number theory is the basis for numerous public-key cryptosystems, digital signature schemes, cryptographic hash functions, and digital cash systems. Every time you buy a book from Amazon, check your grades on WebSIS, or use a PayPal account, you are relying on number theoretic algorithms.

Soon after devising his code, Turing disappeared from public view, and half a century would pass before the world learned the full story of where he'd gone and what he did there. We'll come back to Turing's life in a little while; for now, let's investigate the code Turing left behind. The details are uncertain, since he never formally published the idea, so we'll consider a couple possibilities.

1.1 Turing's Code (Version 1.0)

The first challenge is to translate a text message into an integer so we can perform mathematical operations on it. This step is not intended to make a message harder to read, so the details are not too important. Here is one approach: replace each letter of the message with two digits ($A = 01$, $B = 02$, $C = 03$, etc.) and string all the digits together to form one huge number. For example, the message "victory" could be translated this way:

	"v	i	c	t	o	r	y"
→	22	09	03	20	15	18	25

Turing's code requires the message to be a prime number, so we may need to pad the result with a few more digits to make a prime. In this case, appending the digits 13 gives the number 2209032015182513, which is prime.

Now here is how the encryption process works. In the description below, m is the unencoded message (which we want to keep secret), m^* is the encrypted message (which the Nazis may intercept), and k is the key.

Beforehand The sender and receiver agree on a secret key, which is a large prime k.

Encryption The sender encrypts the message m by computing:

$$m^* = m \cdot k$$

Decryption The receiver decrypts m^* by computing:

$$\frac{m^*}{k} = \frac{m \cdot k}{k} = m$$

For example, suppose that the secret key is the prime number $k = 22801763489$ and the message m is "victory". Then the encrypted message is:

$$\begin{aligned} m^* &= m \cdot k \\ &= 2209032015182513 \cdot 22801763489 \\ &= 50369825549820718594667857 \end{aligned}$$

There are a couple of questions that one might naturally ask about Turing's code.

1. Is Turing's code feasible?
2. Is Turing's code secure?

Let's answer them both at the same time, since they are related: On one hand, the sender and receiver should be able to perform their operations (multiplication and primality testing) efficiently. On the other hand, we want to make sure that an eavesdropper that does not know the secret key *cannot* efficiently factor m!

Well, what do we mean by efficient? We could factor or determine that a number N is prime by trying to divide N by all possible integers less than N. This yields an algorithm that uses something like N operations. These days (in the year 2006) it is typical that N is a 1024 bit number, so we would need 2^{1024} operations. "No problem!" you might say – "we know there must be a factor that is less than $\sqrt{N}$ so just try all possible factors in $1, \ldots, \lceil\sqrt{N}\rceil$!". Unfortunately, this plan only gets us down to 2^{512} operations. This is not totally impossible these days, but it could take a really long time – on the order of several months of work by many computers working together.

Thus, we need our algorithms to be efficient in terms of the *length*, as opposed to the *magnitude* of the integers m and k. For example, we can multiply two numbers in the range $1, \ldots, N$ using time that is grows something like $(\log N)^2$. So, if N is 1024 binary digits, we can multiply in a bit more than 1 million operations. Not fun, but totally doable, especially if you actually care about reading that message.

In addition to multiplying, the sender and receiver both need to verify that k is indeed a prime. The general problem of determining whether a large number is prime or composite has been studied for centuries, and reasonably good primality tests were known even in Turing's time. In 2002, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena gave a primality test that is guaranteed to work on a number N in about $(\log N)^{12}$ steps. This definitively placed primality testing in the class of "easy" computational problems at last. Amazingly, the description of their breakthrough algorithm was only thirteen lines long!

The Nazis see only the encrypted message $m^* = m \cdot k$, so recovering the original message m requires factoring m^*. Despite immense efforts, no really efficient factoring algorithm has ever been found. It appears to be a fundamentally difficult problem, though a breakthrough someday is not impossible. Right now, if you need to factor N in order to

read the message, the best you can do is to use the number field sieve algorithm, which performs $e^{(\frac{6}{4}9N)^{\frac{1}{3}}(\log N)^{\frac{2}{3}}}$ operations. Better get a comfortable chair.

In effect, Turing's code puts to practical use his discovery that there are limits to the power of computation. Thus, provided m and k are sufficiently large, the Nazis seem to be out of luck!

This all sounds promising, but there is a major flaw in Turing's code.

1.2 Breaking Turing's Code

Let's consider what happens when the sender transmits a *second* message using Turing's code and the same key. This gives the Nazis two encrypted messages to look at:

$$m_1^* = m_1 \cdot k \qquad \text{and} \qquad m_2^* = m_2 \cdot k$$

The greatest common divisor of the two encrypted messages, m_1^* and m_2^*, is the secret key k. And, as we've seen, the $\gcd$ of two numbers can be computed very efficiently. So after the second message is sent, the Nazis can read recover the secret key and read *every* message!

It is difficult to believe a mathematician as brilliant as Turing could overlook such a glaring problem. One possible explanation is that he had a slightly different system in mind, one based on *modular* arithmetic.

2 Modular Arithmetic

On page 1 of his masterpiece on number theory, *Disquisitiones Arithmeticae*, Gauss introduced the notion of "congruence". Now, Gauss is another guy who managed to cough up a half-decent idea every now and then, so let's take a look at this one. Gauss said that a ***is congruent to*** b ***modulo*** n if $n \mid (a-b)$. This is denoted $a \equiv b \pmod{n}$. For example:

$$29 \equiv 15 \pmod{7} \quad \text{because } 7 \mid (29-15).$$

Intuitively, the $\equiv$ symbol is sort of like an $=$ sign, and the mod 7 describes the specific sense in which 29 is equal-ish to 15. Thus, even though (mod 7) appears over on the right side, it is in no sense more strongly associated with the 15 than the 29; in fact, it actually defines the meaning of the $\equiv$ sign.

Here's another way to think about congruences: *congruence modulo n defines a partition of the integers into n sets so that congruent numbers are all in the same set*. For example, suppose that we're working modulo 3. Then we can partition the integers into 3 sets as follows:

$$\begin{array}{llllllll} \{ & \ldots, & -6, & -3, & 0, & 3, & 6, & \ldots \} \\ \{ & \ldots, & -5, & -2, & 1, & 4, & 7, & \ldots \} \\ \{ & \ldots, & -4, & -1, & 2, & 5, & 8, & \ldots \} \end{array}$$

Now integers in the same set are all congruent modulo 3. For example, 6 and -3 are both in the first set, and they're congruent because their difference, $6 - (-3) = 9$, is a multiple of 3. Similarly, 11 and 5 are both in the last set, because $11 - 5 = 6$ is a multiple of 3. On the other hand, numbers in different sets are not congruent. For example, 9 is in the first set and 11 in the last set, and they're not congruent because $11 - 9 = 2$ is not a multiple of 3. The upshot is that when arithmetic is done modulo n there are only n really different kinds of number to worry about. In this sense, modular arithmetic is a simplification of ordinary arithmetic and thus is a good reasoning tool.

There are many useful facts about congruences, some of which are listed in the lemma below. The overall theme is that *congruences work a lot like equations*, though there are a couple exceptions.

Lemma 1 (Facts About Congruences)**.** *The following hold for $n \geq 1$:*

1. $a \equiv a \pmod{n}$
2. $a \equiv b \pmod{n}$ *implies* $b \equiv a \pmod{n}$
3. $a \equiv b \pmod{n}$ *and* $b \equiv c \pmod{n}$ *implies* $a \equiv c \pmod{n}$
4. $a \equiv b \pmod{n}$ *implies* $a + c \equiv b + c \pmod{n}$
5. $a \equiv b \pmod{n}$ *implies* $ac \equiv bc \pmod{n}$
6. $a \equiv b \pmod{n}$ *and* $c \equiv d \pmod{n}$ *imply* $a + c \equiv b + d \pmod{n}$
7. $a \equiv b \pmod{n}$ *and* $c \equiv d \pmod{n}$ *imply* $ac \equiv bd \pmod{n}$

Proof. We prove only parts 1 and 7; the other parts are proved similarly.

(part 1) Every integer divides 0, so $n \mid (a - a)$, which means $a \equiv a \pmod{n}$.

(part 7) The assumption $a \equiv b \pmod{n}$ implies that $ac \equiv bc \pmod{n}$ by part 5. Similarly, the assumption $c \equiv d \pmod{n}$ implies $bc \equiv bd \pmod{n}$. Therefore, $ac \equiv bd \pmod{n}$ by part 3. □

There is a close connection between modular arithmetic and the remainder operation, which we looked at last time. To clarify this link, let's reconsider the partition of the integers defined by congruence modulo 3:

$$\begin{array}{llllllll} \{ & \ldots, & -6, & -3, & 0, & 3, & 6, & 9, & \ldots & \} \\ \{ & \ldots, & -5, & -2, & 1, & 4, & 7, & 10, & \ldots & \} \\ \{ & \ldots, & -4, & -1, & 2, & 5, & 8, & 11, & \ldots & \} \end{array}$$

Notice that two numbers are in the same set if and only if they leave the same remainder when divided by 3. The numbers in the first set all leave a remainder of 0 when divided by 3, the numbers in the second set leave a remainder of 1, and the numbers in the third leave a remainder of 2. Furthermore, notice that each number is in the same set as its own remainder. For example, 11 and $\operatorname{rem}(11, 3) = 2$ are both in the same set. Let's bundle all this happy goodness into a lemma.

Lemma 2 (Congruences and Remainders)**.** *The following assertions hold:*

1. $a \equiv (\operatorname{rem}(a, n)) \pmod{n}$
2. $a \equiv b \pmod{n}$ *if and only if* $(\operatorname{rem}(a, n)) = (\operatorname{rem}(b, n))$

Proof. (of part 2) By the division algorithm, there exist unique pairs of integers q_1, r_1 and q_2, r_2 such that:

$$\begin{aligned} a &= q_1 n + r_1 && \text{(where } 0 \leq r_1 < n\text{)} \\ b &= q_2 n + r_2 && \text{(where } 0 \leq r_2 < n\text{)} \end{aligned}$$

In these terms, $(\operatorname{rem}(a, n)) = r_1$ and $(\operatorname{rem}(b, n)) = r_2$. Subtracting the second equation from the first gives:

$$a - b = (q_1 - q_2)n + (r_1 - r_2) \qquad \text{(where } -n < r_1 - r_2 < n\text{)}$$

Now $a \equiv b \pmod{n}$ if and only if n divides the left side. This is true if and only if n divides the right side, which holds if and only if $r_1 - r_2$ is a multiple of n. Given the bounds on $r_1 = r_2$, this happens precisely when $r_1 = r_2$, which is equivalent to $\operatorname{rem}(a, n)) = (\operatorname{rem}(b, n))$. □

3 Turing's Code (Version 2.0)

In 1940 France had fallen before Hitler's army, and Britain alone stood against the Nazis in western Europe. British resistance depended on a steady flow of supplies brought across the north Atlantic from the United States by convoys of ships. These convoys were engaged in a cat-and-mouse game with German "U-boat" submarines, which prowled the Atlantic, trying to sink supply ships and starve Britain into submission. The outcome of this struggle pivoted on a balance of information: could the Germans locate convoys better than the Allies could locate U-boats or vice versa?

Germany lost.

But a critical reason behind Germany's loss was made public only in 1974: the British had broken Germany's naval code, Enigma. Through much of the war, the Allies were able to route convoys around German submarines by listening into German communications. The British government didn't explain *how* Enigma was broken until 1996. When the analysis was finally released (by the US), the author was none other than Alan Turing. In 1939 he had joined the secret British codebreaking effort at Bletchley Park. There, he played a central role in cracking the German's Enigma code and thus in preventing Britain from falling into Hitler's hands.

Governments are always tight-lipped about cryptography, but the half-century of official silence about Turing's role in breaking Enigma and saving Britain may be related to some disturbing events after the war.

Let's consider an alternative interpretation of Turing's code. Perhaps we had the basic idea right (multiply the message by the key), but erred in using *conventional* arithmetic instead of *modular* arithemtic. Maybe this is what Turing meant:

Beforehand The sender and receiver agree on a large prime p, which may be made public. (This will be the modulus for all our arithemtic.) They also agree on a secret key $k \in \{1, 2, \ldots, p-1\}$.

Encryption The message m can be any integer in the set $\{0, 1, 2, \ldots, p-1\}$; in particular, the message is no longer required to be a prime. The sender encrypts the message m to produce m^* by computing:

$$m^* = \text{rem}\,(mk, p) \qquad (*)$$

Decryption (Uh-oh.)

The decryption step is a problem. We might hope to decrypt in the same way as before: by dividing the encrypted message m^* by the key k. The difficulty is that $m*$ is the *remainder* when mk is divided by p. So dividing m^* by k might not even give us an integer!

This decoding difficulty can be overcome with a better understanding of arithmetic modulo a prime.

3.1 Multiplicative Inverses

The ***mutiplicative inverse*** of a number x is another number x^{-1} such that:

$$x \cdot x^{-1} = 1$$

Generally, multiplicative inverses exist over the real numbers. For example, the multiplicative inverse of 3 is $1/3$ since:

$$3 \cdot \frac{1}{3} = 1$$

The sole exception is that 0 does not have an inverse.

On the other hand, inverses generally do not exist over the integers. For example, 7 can not be multiplied by another integer to give 1.

Surprisingly, multiplicative inverses do exist when we're working *modulo a prime number*. For example, if we're working modulo 5, then 3 is a multiplicative inverse of 7, since:

$$7 \cdot 3 \equiv 1 \pmod 5$$

(All numbers congruent to 3 modulo 5 are also multiplicative inverses of 7; for example, $7 \cdot 8 \equiv 1 \pmod 5$ as well.) The only exception is that numbers congruent to 0 modulo 5 (that is, the multiples of 5) do not have inverses, much as 0 does not have an inverse over the real numbers. Let's prove this.

Lemma 3. *If p is prime and k is not a multiple of p, then k has a multiplicative inverse modulo p.*

Proof. Since p is prime, it has only two divisors: 1 and p. And since k is not a multiple of p, we must have $\gcd(p, k) = 1$. Therefore, there is a linear combination of p and k equal to 1:

$$sp + tk = 1$$

Rearranging terms gives:

$$sp = 1 - tk$$

This implies that $p \mid 1 - tk$ by the definition of divisibility, and therefore $tk \equiv 1 \pmod{p}$ by the definition of congruence. Thus, t is a multiplicative inverse of k. □

Multiplicative inverses are the key to decryption in Turing's code. Specifically, we can recover the original message by multiplying the encoded message by the *inverse* of the key:

$$\begin{aligned} m^* \cdot k^{-1} &\equiv \operatorname{rem}(mk, p) \cdot k^{-1} \pmod{p} \\ &\equiv mkk^{-1} \pmod{p} \\ &\equiv m \pmod{p} \end{aligned}$$

This shows that m^*k^{-1} is congruent to the original message m. Since the m was in the range $0, 1, \ldots, p-1$, we can recover it exactly taking a remainder:

$$m = \operatorname{rem}(m^*k^{-1}, p)$$

So now we can decrypt!

3.2 Cancellation

Another sense in which real number are nice is that one can cancel multiplicative terms. In other words, if we know that $m_1k = m_2k$, then can cancel the k's and conclude that $m_1 = m_2$, provided $k \neq 0$. In general, cancellation is *not* valid in modular arithmetic. For example, this congruence is correct:

$$2 \cdot 3 \equiv 4 \cdot 3 \pmod{6}$$

But if we cancel the 3's, we reach a false conclusion:

$$2 \equiv 4 \pmod{6}$$

The fact that multiplicative terms can not be cancelled is the most significant sense in which congruences differ from ordinary equations. However, this difference goes away if we're working modulo a *prime*; then cancellation is valid.

Lemma 4. *Suppose p is a prime and k is not a multiple of p. Then*

$$ak \equiv bk \pmod{p} \qquad \textit{implies} \qquad a \equiv b \pmod{p}$$

Proof. Multiply both sides of the congruence by k^{-1}. □

We can use this lemma to get a bit more insight into how Turing's code works. In particular, the encryption operation in Turing's code *permutes the space of messages*. This is stated more precisely in the following corollary.

Corollary 5. *Suppose p is a prime and k is not a multiple of p. Then the sequence:*

$$\text{rem}\,((0 \cdot k), p), \quad \text{rem}\,((1 \cdot k), p), \quad \text{rem}\,((2 \cdot k), p), \quad \ldots, \quad \text{rem}\,(\big((p-1) \cdot k\big), p)$$

is a permutation of the sequence:

$$0, \quad 1, \quad 2, \quad \ldots, \quad (p-1)$$

This remains true if the first term is deleted from each sequence.

Proof. The first sequence contains p numbers, which are all in the range 0 to $p-1$ by the definition of remainder. Furthermore, the numbers in the first sequence are all different; by part 2 of Lemma 2, $\text{rem}\,(m_1 k, p) = \text{rem}\,(m_2 k, p)$ if and only if $m_1 \equiv m_2 \pmod{p}$, and no two numbers in the range 0, 1, . . . , p - 1 are congruent modulo p. Thus, the first sequence must contain *all* of the numbers from 0 to $p-1$ in some order. The claim remains true if the first terms are deleted, because both sequences begin with 0. □

For example, suppose $p = 5$ and $k = 3$. Then the sequence:

$$\underbrace{\text{rem}\,((0 \cdot 3), 5)}_{=0}, \quad \underbrace{\text{rem}\,((1 \cdot 3), 5)}_{=3}, \quad \underbrace{\text{rem}\,((2 \cdot 3), 5)}_{=1}, \quad \underbrace{\text{rem}\,((3 \cdot 3), 5)}_{=4}, \quad \underbrace{\text{rem}\,((4 \cdot 3), 5)}_{=2}$$

is a permutation of 0, 1, 2, 3, 4 and the last four terms are a permutation of 1, 2, 3, 4. As long as the Nazis don't know the secret key k, they don't know how the message space is permuted by the process of encryption and thus can't read encoded messages.

3.3 Fermat's Theorem

A remaining challenge in using Turing's code is that decryption requires the inverse of the secret key k. But how can we find an inverse of k? One approach is to rely on Fermat's Theorem, which is much easier than his famous Last Theorem— and more useful.

Theorem 6 (Fermat's Theorem)**.** *Suppose p is a prime and k is not a multiple of p. Then:*

$$k^{p-1} \equiv 1 \pmod{p}$$

Proof. We reason as follows:

$$\begin{aligned} 1 \cdot 2 \cdot 3 \cdots (p-1) &\equiv \operatorname{rem}(k,p) \cdot \operatorname{rem}(2k,p) \cdot \operatorname{rem}(3k,p) \cdots \operatorname{rem}((p-1)k,p) \pmod{p} \\ &\equiv k \cdot 2k \cdot 3k \cdots (p-1)k \pmod{p} \\ &\equiv (p-1)! \cdot k^{p-1} \pmod{p} \end{aligned}$$

The expressions on the first line are actually equal, by Corollary 5, so they are certainly congruent modulo p. The second step uses part 1 of Lemma 2. In the third step, we rearrange terms in the product.

Now $(p-1)!$ can not be a multiple of p, because the prime factorizations of $1, 2, \ldots, (p-1)$ contain only primes smaller than p. Therefore, we can cancel $(p-1)!$ from the first expression and the last by Lemma 4, which proves the claim. □

Here is how we can find inverses using Fermat's Theorem. Suppose p is a prime and k is not a multiple of p. Then, by Fermat's Theorem, we know that:

$$k^{p-2} \cdot k \equiv 1 \pmod{p}$$

Therefore, k^{p-2} must be a multiplicative inverse of k. For example, suppose that we want the multiplicative inverse of 6 modulo 17. Then we need to compute $\operatorname{rem}(6^{15}, 17)$, which we can do by successive squaring. All the congruences below hold modulo 17.

$$\begin{aligned} 6^2 &\equiv 36 \equiv 2 \\ 6^4 &\equiv (6^2)^2 \equiv 2^2 \equiv 4 \\ 6^8 &\equiv (6^4)^2 \equiv 4^2 \equiv 16 \\ 6^{15} &\equiv 6^8 \cdot 6^4 \cdot 6^2 \cdot 6 \equiv 16 \cdot 4 \cdot 2 \cdot 6 \equiv 3 \end{aligned}$$

Therefore, $\operatorname{rem}(6^{15}, 17) = 3$. Sure enough, 3 is the multiplicative inverse of 6 modulo 17, since:

$$3 \cdot 6 \equiv 1 \pmod{17}$$

In general, if we were working modulo a prime p, finding a multiplicative inverse by trying every value between 1 and $p-1$ would require about p operations. However, the approach above requires only about $\log p$ operations, which is far better when p is large.

3.4 Breaking Turing's Code— Again

German weather reports were *not* encrypted with the highly-secure Enigma system. After all, so what if the Allies learned that there was rain off the south coast of Iceland? But, amazingly, this practice provided the British with a critical edge in the Atlantic naval battle during 1941.

The problem was that some of those weather reports had originally been transmitted from U-boats out in the Atlantic. Thus, the British obtained both unencrypted reports and the same reports encrypted with Enigma. By comparing the two, the British were able to determine which key the Germans were using that day and could read all other Enigma-encoded traffic. Today, this would be called a ***known-plaintext attack***.

Let's see how a known-plaintext attack would work against Turing's code. Suppose that the Nazis know both m and m^* where:

$$m^* \equiv mk \pmod{p}$$

Now they can compute:

$$\begin{aligned} m^{p-2} \cdot m^* &\equiv m^{p-2} \cdot \operatorname{rem}(mk, p) \pmod{p} && \text{(def. of } m^*\text{)} \\ &\equiv m^{p-2} \cdot mk \pmod{p} && \text{(part 2 of Lemma 2)} \\ &\equiv m^{p-1} \cdot k \pmod{p} && \text{(simplification)} \\ &\equiv k \pmod{p} && \text{(Fermat's Theorem)} \end{aligned}$$

Now the Nazis have the secret key k and can decrypt any message!

This is a huge vulnerability, so Turing's code has no practical value. Fortunately, Turing got better at cryptography after devising this code; his subsequent cracking of Enigma surely saved thousands of lives, if not the whole of Britain.

4 Postscript

A few years after the war, Turing's home was robbed. Detectives soon determined that a former homosexual lover of Turing's had conspired in the robbery. So they arrested him; that is, they arrested Alan Turing. Because, at that time, homosexuality was a crime in Britain, punishable by up to two years in prison. Turing was sentenced to a humiliating hormonal "treatment" for his homosexuality: he was given estrogen injections. He began to develop breasts.

Three years later, Alan Turing, the founder of computer science, was dead. His mother explained what happened in a biography of her own son. Despite her repeated warnings, Turing carried out chemistry experiments in his own home. Apparently, her worst fear was realized: by working with potassium cyanide while eating an apple, he poisoned himself.

However, Turing remained a puzzle to the very end. His mother was a devoutly religious woman who considered suicide a sin. And, other biographers have pointed out, Turing had previously discussed committing suicide by eating a poisoned apple. Evidently, Alan Turing, who founded computer science and saved his country, took his own life in the end, and in just such a way that his mother could believe it was an accident.