CPSC 467: Cryptography and Computer Security

Michael J. Fischer

Lecture 9
September 25, 2013

Integer Division

- Quotient, remainder, and mod
- The mod relation
- GCD
- Relatively prime numbers, $\mathbf{Z}_n^*$, and $\phi(n)$

Discrete Logarithm

Diffie-Hellman Key Exchange

ElGamal Key Agreement

Integer Division

Quotient and remainder

Theorem (division theorem)

Let a, b be integers and assume $b > 0$. There are unique integers q (the quotient) and r (the remainder) such that $a = bq + r$ and $0 \leq r < b$.

Write the quotient as $a \div b$ and the remainder as $a \bmod b$. Then

$$a = b \times (a \div b) + (a \bmod b).$$

Equivalently,

$$a \bmod b = a - b \times (a \div b).$$

$$a \div b = \lfloor a/b \rfloor.^{1}$$

[1] Here, / is ordinary real division and $\lfloor x \rfloor$, the *floor* of x, is the greatest integer $\leq x$. In C, / is used for both $\div$ and / depending on its operand types.

The mod operator for negative numbers

When either a or b is negative, there is no consensus on the definition of $a \bmod b$.

By our definition, $a \bmod b$ is always in the range $[0 \ldots b-1]$, even when a is negative.

Example,

$$(-5) \bmod 3 = (-5) - 3 \times ((-5) \div 3) = -5 - 3 \times (-2) = 1.$$

The mod operator % in C

In the C programming language, the mod operator % is defined differently, so $(a \,\%\, b) \neq (a \bmod b)$ when a is negative and b is positive.

The C standard defines $a \,\%\, b$ to be the number r satisfying the equation $(a/b) * b + r = a$, so $r = a - (a/b) * b$.

C also defines a/b to be the result of rounding the real number a/b towards zero, so $-5/3 = -1$. Hence,

$$-5 \,\%\, 3 = -5 - (-5/3) * 3 = -5 + 3 = -2.$$

Divides

We say that b *divides* a (exactly) and write $b \mid a$ in case $a \bmod b = 0$.

Fact

If $d \mid (a+b)$, then either d divides both a and b, or d divides neither of them.

To see this, suppose $d \mid (a+b)$ and $d \mid a$. Then by the division theorem, $a+b = dq_1$ and $a = dq_2$ for some integers q_1 and q_2. Substituting for a and solving for b, we get

$$b = dq_1 - dq_2 = d(q_1 - q_2).$$

But this implies $d \mid b$, again by the division theorem.

The mod relation

We just saw that mod is a binary operation on integers.

Mod is also used to denote a relationship on integers:

$$a \equiv b \pmod{n} \quad \text{iff} \quad n \mid (a - b).$$

That is, a and b have the same remainder when divided by n. An immediate consequence of this definition is that

$$a \equiv b \pmod{n} \quad \text{iff} \quad (a \bmod n) = (b \bmod n).$$

Thus, the two notions of mod aren't so different after all!

We sometimes write $a \equiv_n b$ to mean $a \equiv b \pmod{n}$.

Mod is an equivalence relation

The two-place relationship $\equiv_n$ is an *equivalence relation*.

Its equivalence classes are called *residue* classes modulo n and are denoted by $[b]_{\equiv_n} = \{a \mid a \equiv b \pmod{n}\}$ or simply by $[b]$.

For example, if $n = 7$, then $[10] = \{\ldots - 11, -4, 3, 10, 17, \ldots\}$.

Fact

$[a] = [b]$ *iff* $a \equiv b \pmod{n}$.

Canonical names

If $x \in [b]$, then x is said to be a *representative* or *name* of the equivalence class $[b]$. Obviously, b is a representative of $[b]$. Thus, $[-11]$, $[-4]$, $[3]$, $[10]$, $[17]$ are all names for the same equivalence class.

The *canonical* or preferred name for the class $[b]$ is the unique integer in $[b] \cap \{0, 1, \ldots, n-1\}$.

Thus, the canonical name for $[10]$ is $10 \bmod 7 = 3$.

Mod is a congruence relation

The relation $\equiv_n$ is a *congruence relation* with respect to addition, subtraction, and multiplication of integers.

Fact

For each arithmetic operation $\odot \in \{+, -, \times\}$, if $a \equiv a' \pmod{n}$ and $b \equiv b' \pmod{n}$, then

$$a \odot b \equiv a' \odot b' \pmod{n}.$$

The class containing the result of $a \odot b$ depends only on the classes to which a and b belong and not the particular representatives chosen.

Hence, we can perform arithmetic on equivalence classes by operating on their names.

Greatest common divisor

Definition

The *greatest common divisor* of two integers a and b, written $\gcd(a,b)$, is the largest integer d such that $d \mid a$ and $d \mid b$.

$\gcd(a,b)$ is always defined unless $a = b = 0$ since 1 is a divisor of every integer, and the divisor of a non-zero number cannot be larger (in absolute value) than the number itself.

Question: Why isn't $\gcd(0,0)$ well defined?

Computing the GCD

$\gcd(a, b)$ is easily computed if a and b are given in factored form.

Namely, let p_i be the i^{th} prime. Write $a = \prod p_i^{e_i}$ and $b = \prod p_i^{f_i}$. Then

$$\gcd(a, b) = \prod p_i^{\min(e_i, f_i)}.$$

Example: $168 = 2^3 \cdot 3 \cdot 7$ and $450 = 2 \cdot 3^2 \cdot 5^2$, so $\gcd(168, 450) = 2 \cdot 3 = 6$.

However, factoring is believed to be a hard problem, and no polynomial-time factorization algorithm is currently known. (If it were easy, then Eve could use it to break RSA, and RSA would be of no interest as a cryptosystem.)

Euclidean algorithm

Fortunately, $\gcd(a, b)$ can be computed efficiently without the need to factor a and b using the famous *Euclidean algorithm*.

Euclid's algorithm is remarkable, not only because it was discovered a very long time ago, but also because it works without knowing the factorization of a and b.

Euclidean identities

The Euclidean algorithm relies on several identities satisfied by the gcd function. In the following, assume $a > 0$ and $a \geq b \geq 0$:

$$\gcd(a, b) = \gcd(b, a) \tag{1}$$
$$\gcd(a, 0) = a \tag{2}$$
$$\gcd(a, b) = \gcd(a - b, b) \tag{3}$$

Identity 1 is obvious from the definition of gcd. Identity 2 follows from the fact that every positive integer divides 0. Identity 3 follows from the basic fact relating divides and addition on slide 7.

Computing GCD without factoring

The Euclidean identities allow the problem of computing $\gcd(a, b)$ to be reduced to the problem of computing $\gcd(a - b, b)$.

The new problem is "smaller" as long as $b > 0$.

The *size* of the problem $\gcd(a, b)$ is $|a| + |b|$, the sum of the two arguments. This leads to an easy recursive algorithm.

```
int gcd(int a, int b)
{
  if ( a < b ) return gcd(b, a);
  else if ( b == 0 ) return a;
  else return gcd(a-b, b);
}
```

Nevertheless, this algorithm is not very efficient, as you will quickly discover if you attempt to use it, say, to compute $\gcd(1000000, 2)$.

Repeated subtraction

Repeatedly applying identity (3) to the pair (a, b) until it can't be applied any more produces the sequence of pairs

$$(a, b), (a - b, b), (a - 2b, b), \ldots, (a - qb, b).$$

The sequence stops when $a - qb < b$.

How many times you can subtract b from a while remaining non-negative?
Answer: The quotient $q = \lfloor a/b \rfloor$.

Using division in place of repeated subtractions

The amout $a - qb$ that is left after q subtractions is just the remainder $a \bmod b$.

Hence, one can go directly from the pair (a, b) to the pair $(a \bmod b, b)$.

This proves the identity

$$\gcd(a, b) = \gcd(a \bmod b, b). \tag{4}$$

Full Euclidean algorithm

Recall the inefficient GCD algorithm.

```
int gcd(int a, int b) {
  if ( a < b ) return gcd(b, a);
  else if ( b == 0 ) return a;
  else return gcd(a-b, b);
}
```

The following algorithm is exponentially faster.

```
int gcd(int a, int b) {
  if ( b == 0 ) return a;
  else return gcd(b, a%b);
}
```

Principal change: Replace `gcd(a-b,b)` with `gcd(b, a%b)`.

Besides collapsing repeated subtractions, we have $a \geq b$ for all but the top-level call on $\gcd(a, b)$. This eliminates roughly half of the remaining recursive calls.

Complexity of GCD

The new algorithm requires at most in $O(n)$ stages, where n is the sum of the lengths of a and b when written in binary notation, and each stage involves at most one remainder computation.

The following iterative version eliminates the stack overhead:

```
int gcd(int a, int b) {
  int aa;
  while (b > 0) {
    aa = a;
    a = b;
    b = aa % b;
  }
  return a;
}
```

Relatively prime numbers

Two integers a and b are *relatively prime* if they have no common prime factors.

Equivalently, a and b are *relatively prime* if $\gcd(a, b) = 1$.

Let $\mathbf{Z}_n^*$ be the set of integers in $\mathbf{Z}_n$ that are relatively prime to n, so

$$\mathbf{Z}_n^* = \{a \in \mathbf{Z}_n \mid \gcd(a, n) = 1\}.$$

Euler's totient function $\phi(n)$

$\phi(n)$ is the cardinality (number of elements) of $\mathbf{Z}_n^*$, i.e.,

$$\phi(n) = |\mathbf{Z}_n^*|.$$

Properties of $\phi(n)$:

1. If p is prime, then
$$\phi(p) = p - 1.$$
2. More generally, if p is prime and $k \geq 1$, then
$$\phi(p^k) = p^k - p^{k-1} = (p-1)p^{k-1}.$$
3. If $\gcd(m, n) = 1$, then
$$\phi(mn) = \phi(m)\phi(n).$$

Example: $\phi(26)$

Can compute $\phi(n)$ for all $n \geq 1$ given the factorization of n.

$$\begin{aligned}\phi(126) &= \phi(2) \cdot \phi(3^2) \cdot \phi(7) \\ &= (2-1) \cdot (3-1)(3^{2-1}) \cdot (7-1) \\ &= 1 \cdot 2 \cdot 3 \cdot 6 = 36.\end{aligned}$$

The 36 elements of $\mathbf{Z}_{126}^*$ are:

1, 5, 11, 13, 17, 19, 23, 25, 29, 31, 37, 41, 43, 47, 53, 55, 59, 61, 65, 67, 71, 73, 79, 83, 85, 89, 95, 97, 101, 103, 107, 109, 113, 115, 121, 125.

A formula for $\phi(n)$

Here is an explicit formula for $\phi(n)$.

Theorem

Write n in factored form, so $n = p_1^{e_1} \cdots p_k^{e_k}$, where $p_1, \ldots, p_k$ are distinct primes and $e_1, \ldots, e_k$ are positive integers.[2] *Then*

$$\phi(n) = (p_1 - 1) \cdot p_1^{e_1 - 1} \cdots (p_k - 1) \cdot p_k^{e_k - 1}.$$

For the product of distinct primes p and q,

$$\phi(pq) = (p-1)(q-1).$$

[2] By the fundamental theorem of arithmetic, every integer can be written uniquely in this way up to the ordering of the factors.

Discrete Logarithm

Logarithms mod p

Let $y = b^x$ over the reals. The ordinary base-b logarithm is the inverse of the exponential function, so $x = \log_b(y)$

The discrete logarithm is defined similarly, but now arithmetic is performed in $\mathbf{Z}_p^*$ for a prime p.

In particular, the base-b *discrete logarithm* of y modulo p is the least non-negative integer x such that $y \equiv b^x \pmod{p}$ (if it exists). We write $x = \log_b(y) \bmod p$.

Fact (not needed yet): If b is a *primitive root*[3] of p, then $\log_b(y)$ is defined for every $y \in \mathbf{Z}_p^*$.

[3] We will talk about primitive roots later.

Discrete log problem

The *discrete log problem* is the problem of computing $\log_b(y) \bmod p$, where p is a prime and b is a primitive root of p.

No efficient algorithm is known for this problem and it is believed to be intractable.

However, the inverse of the function $\log_b() \bmod p$ is the function $\text{power}_b(x) = b^x \bmod p$, which is easily computable.

power_b is believed to be a *one-way function*, that is a function that is easy to compute but hard to invert.

Diffie-Hellman Key Exchange

Key exchange problem

The *key exchange problem* is for Alice and Bob to agree on a common random key k.

One way for this to happen is for Alice to choose k at random and then communicate it to Bob over a secure channel.

But that presupposes the existence of a secure channel.

D-H key exchange overview

The Diffie-Hellman Key Exchange protocol allows Alice and Bob to agree on a secret k without having prior secret information and without giving an eavesdropper Eve any information about k. The protocol is given on the next slide.

We assume that p and g are publicly known, where p is a large prime and g a primitive root of p.

D-H key exchange protocol

Alice	**Bob**
Choose random $x \in \mathbf{Z}_{\phi(p)}$.	Choose random $y \in \mathbf{Z}_{\phi(p)}$.
$a = g^x \bmod p$.	$b = g^y \bmod p$.
Send a to Bob.	Send b to Alice.
$k_a = b^x \bmod p$.	$k_b = a^y \bmod p$.

Diffie-Hellman Key Exchange Protocol.

Clearly, $k_a = k_b$ since

$$k_a \equiv b^x \equiv g^{xy} \equiv a^y \equiv k_b \pmod{p}.$$

Hence, $k = k_a = k_b$ is a common key.

Security of DH key exchange

In practice, Alice and Bob can use this protocol to generate a session key for a symmetric cryptosystem, which they can subsequently use to exchange private information.

The security of this protocol relies on Eve's presumed inability to compute k from a and b and the public information p and g. This is sometime called the *Diffie-Hellman problem* and, like discrete log, is believed to be intractable.

Certainly the Diffie-Hellman problem is no harder that discrete log, for if Eve could find the discrete log of a, then she would know x and could compute k_a the same way that Alice does.

However, it is not known to be as hard as discrete log.

ElGamal Key Agreement

A variant of DH key exchange

A variant protocol has Bob going first followed by Alice.

Alice	**Bob**
	Choose random $y \in \mathbf{Z}_{\phi(p)}$.
	$b = g^y \bmod p$.
	Send b to Alice.
Choose random $x \in \mathbf{Z}_{\phi(p)}$.	
$a = g^x \bmod p$.	
Send a to Bob.	
$k_a = b^x \bmod p$.	$k_b = a^y \bmod p$.

ElGamal Variant of Diffie-Hellman Key Exchange.

Comparison with first DH protocol

The difference here is that Bob completes his action at the beginning and no longer has to communicate with Alice.

Alice, at a later time, can complete her half of the protocol and send a to Bob, at which point Alice and Bob share a key.

This is just the scenario we want for public key cryptography. Bob generates a public key (p, g, b) and a private key (p, g, y).

Alice (or anyone who obtains Bob's public key) can complete the protocol by sending a to Bob.

This is the idea behind the ElGamal public key cryptosystem.

ElGamal cryptosystem

Assume Alice knows Bob's public key (p, g, b). To encrypt a message m:

- She first completes her part of the key exchange protocol to obtain numbers a and k.
- She then computes $c = mk \bmod p$ and sends the pair (a, c) to Bob.
- When Bob gets this message, he first uses a to complete his part of the protocol and obtain k.
- He then computes $m = k^{-1}c \bmod p$.

Combining key exchange with underlying cryptosystem

The ElGamal cryptosystem uses the simple encryption function $E_k(m) = mk \bmod p$ to actually encode the message.

Any symmetric cryptosystem would work equally well.

An advantage of using a standard system such as AES is that long messages can be sent following only a single key exchange.

A hybrid ElGamal cryptosystem

A hybrid ElGamal public key cryptosystem.

- As before, Bob generates a public key (p, g, b) and a private key (p, g, y).
- To encrypt a message m to Bob, Alice first obtains Bob's public key and chooses a random $x \in \mathbf{Z}_{\phi(p)}$.
- She next computes $a = g^x \bmod p$ and $k = b^x \bmod p$.
- She then computes $E_{(p,g,b)}(m) = (a, \hat{E}_k(m))$ and sends it to Bob. Here, $\hat{E}$ is the encryption function of the underlying symmetric cryptosystem.
- Bob receives a pair (a, c).
- To decrypt, Bob computes $k = a^y \bmod p$ and then computes $m = \hat{D}_k(c)$.

Randomized encryption

We remark that a new element has been snuck in here. The ElGamal cryptosystem and its variants require Alice to generate a random number which is then used in the course of encryption.

Thus, the resulting encryption function is a *random function* rather than an ordinary function.

A random function is one that can return different values each time it is called, even for the same arguments.

Formally, we view a random function as returning a probability distribution on the output space.

Remarks about randomized encryption

With $E_{(p,g,b)}(m)$ each message m has many different possible encryptions. This has some consequences.

An advantage: Eve can no longer use the public encryption function to check a possible decryption.

Even if she knows m, she cannot verify m is the correct decryption of (a, c) simply by computing $E_{(p,g,b)}(m)$, which she could do for a deterministic cryptosystem such as RSA.

Two disadvantages:

- Alice must have a source of randomness.
- The ciphertext is longer than the corresponding plaintext.