CPSC 467b: Cryptography and Computer Security

Michael J. Fischer

Lecture 8
February 7, 2013

Fast Exponentiation Algorithms

Number Theory Needed for RSA

Integer Division

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

Fast Exponentiation Algorithms

Modular exponentiation

The basic operation of RSA is modular exponentiation of big numbers, i.e., computing $m^e \bmod n$ for big numbers m, e, and n.

The obvious way to compute this would be to compute first $t = m^e$ and then compute $t \bmod n$.

This has two serious drawbacks.

Computing m^e the conventional way is too slow

The simple iterative loop to compute m^e requires e multiplications, or about 2^{1024} operations in all. This computation would run longer than the current age of the universe (which is estimated to be 15 billion years).

Assuming one loop iteration could be done in one microsecond (very optimistic seeing as each iteration requires computing a product and remainder of big numbers), only about 30×10^{12} iterations could be performed per year, and only about 450×10^{21} iterations in the lifetime of the universe. But $450 \times 10^{21} \approx 2^{79}$, far less than $e - 1$.

The result of computing m^e is too big to write down.

The number m^e is too big to store! This number, when written in binary, is about $1024 * 2^{1024}$ bits long, a number far larger than the number of atoms in the universe (which is estimated to be only around $10^{80} \approx 2^{266}$).

Controlling the size of intermediate results

The trick to get around the second problem is to do all arithmetic modulo n, that is, reduce the result modulo n after each arithmetic operation.

The product of two length ℓ numbers is only length 2ℓ before reduction mod n, so in this way, one never has to deal with numbers longer than about 2048 bits.

Question to think about: Why is it correct to do this?

Efficient exponentiation

The trick to avoiding the first problem is to use a more efficient exponentiation algorithm based on repeated squaring.

For the special case of $e = 2^k$, one computes $m^e \bmod n$ as follows:

$$\begin{aligned} m_0 &= m \\ m_1 &= (m_0 * m_0) \bmod n \\ m_2 &= (m_1 * m_1) \bmod n \\ &\vdots \\ m_k &= (m_{k-1} * m_{k-1}) \bmod n. \end{aligned}$$

Clearly, $m_i = m^{2^i} \bmod n$ for all i.

Combining the m_i for general e

For values of e that are not powers of 2, $m^e \bmod n$ can be obtained as the product modulo n of certain m_i's.

Express e in binary as $e = (b_s b_{s-1} \ldots b_2 b_1 b_0)_2$. Then $e = \sum_i b_i 2^i$, so

$$m^e = m^{\sum_i b_i 2^i} = \prod_i m^{b_i 2^i} = \prod_i (m^{2^i})^{b_i} = \prod_{i:\ b_I = 1} m_i.$$

Since each $b_i \in \{0, 1\}$, we include exactly those m_i in the final product for which $b_i = 1$. Hence,

$$m^e \bmod n = \prod_{i:\ b_I = 1} m_i \bmod n.$$

Towards greater efficiency

It is not necessary to perform this computation in two phases.

Rather, the two phases can be combined together, resulting in slicker and simpler algorithms that do not require the explicit storage of the m_i's.

We give both a recursive and an iterative version.

A recursive exponentiation algorithm

Here is a recursive version written in C notation, but it should be understood that the C programs only work for numbers smaller than 2^{16}. To handle larger numbers requires the use of big number functions.

```
/* computes m^e mod n recursively */
int modexp( int m, int e, int n) {
  int r;
  if ( e == 0 ) return 1;          /* m^0 = 1 */
  r = modexp(m*m % n, e/2, n);     /* r = (m^2)^(e/2) mod n */
  if ( (e&1) == 1 ) r = r*m % n;   /* handle case of odd e */
  return r;
}
```

An iterative exponentiation algorithm

This same idea can be expressed iteratively to achieve even greater efficiency.

```
/* computes m^e mod n iteratively */
int modexp( int m, int e, int n) {
  int r = 1;
  while ( e > 0 ) {
    if ( (e&1) == 1 ) r = r*m % n;
    e /= 2;
    m = m*m % n;
  }
  return r;
}
```

Correctness

The loop invariant is

$$e > 0 \wedge (m_0^{e_0} \bmod n = rm^e \bmod n) \tag{1}$$

where m_0 and e_0 are the initial values of m and e, respectively.

Proof of correctness:

- It is easily checked that (1) holds at the start of each iteration.
- If the loop exits, then $e = 0$, so $r \bmod n$ is the desired result.
- Termination is ensured since e gets reduced during each iteration.

A minor optimization

Note that the last iteration of the loop computes a new value of m that is never used. A slight efficiency improvement results from restructuring the code to eliminate this unnecessary computation. Following is one way of doing so.

```
/* computes m^e mod n iteratively */
int modexp( int m, int e, int n) {
  int r = ( (e&1) == 1 ) ? m % n : 1;
  e /= 2;
  while ( e > 0 ) {
    m = m*m % n;
    if ( (e&1) == 1 ) r = r*m % n;
    e /= 2;
  }
  return r;
}
```

Number Theory Needed for RSA

Number theory overview

In this and following sections, we review some number theory that is needed for understanding RSA.

I will provide only a high-level overview. Further details are contained in course handouts and the textbooks.

Summary of what is needed

Here's a summary of the number theory needed to understand RSA and its associate algorithms.

- Greatest common divisor, $\mathbf{Z}_n$, mod n, $\phi(n)$, $\mathbf{Z}_n^*$, and how to add, subtract, multiply, and find inverses mod n.
- Euler's theorem: $a^{\phi(n)} \equiv 1 \pmod{n}$ for $a \in \mathbf{Z}_n^*$.
- How to generate large prime numbers: density of primes and testing primality.

How these facts apply to RSA

- The RSA key pair (e, d) is chosen to satisfy the *modular equation* $ed \equiv 1 \pmod{\phi(n)}$.
- To find (e, d), we repeatedly choose e at random from $\mathbf{Z}_n$ until we find one in $\mathbf{Z}_n^*$, and then *solve* the modular equation $ed \equiv 1 \pmod{\phi(n)}$ for d. We compute gcd to test for membership in $\mathbf{Z}_n^*$.
- Using *Euler's theorem*, we can show $m^{ed} \equiv m \pmod{n}$ for all $m \in \mathbf{Z}_n^*$. This implies $D_d(E_e(m)) = m$. To show that decryption works even in the rare case that $m \in \mathbf{Z}_n - \mathbf{Z}_n^*$ requires some more number theory that we will omit.
- To find p and q, we choose large numbers and *test each for primality* until we find two distinct primes. We must show that the *density of primes* is large enough for this procedure to be feasible.

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 $a \geq b \geq 0$:

$$\gcd(a, b) = \gcd(b, a) \quad (2)$$
$$\gcd(a, 0) = a \quad (3)$$
$$\gcd(a, b) = \gcd(a - b, b) \quad (4)$$

Identity 2 is obvious from the definition of gcd. Identity 3 follows from the fact that every positive integer divides 0. Identity 4 follows from the basic fact relating divides and addition from lecture 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 (4) 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

$$\gcd(a, b) = \gcd(a \bmod b, b). \tag{5}$$

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
$$\phi(mn) = \phi(m)\phi(n).$$